Approfondimenti tematici

Android PROGRAMMING

Il corso completo per imparare a programmare con il S.O. Google dedicato agli smartphone

Android programming

Questo approfondimento tematico è pensato per chi vuol imparare a programmare e creare software per gli smartphone con sistema operativo Google Android. La prima parte del testo guida il lettore alla conoscenza degli strumenti necessari per sviluppare sulla piattaforma mobile di Mountain View (installazione SDK, librerie e tool di supporto allo sviluppo).
Le sezioni successive sono pensate per un apprendimento pratico basato su esempi di progetto: dialogo e interazione con l'ambiente operativo del telefonino, interazione con gli utenti, componenti di un widget, interfacce in XML, gestione del touch, progettazione dei menu e via dicendo.
Una serie di esempi pratici da seguire passo passo che spingono il lettore a sperimentare sul campo il proprio livello di apprendimento e lo invitano a imparare divertendosi.

PROGRAMMARE GOOGLE ANDROID

PRIMO APPUNTAMENTO ALLA SCOPERTA DI ANDROID. INSTALLEREMO GLI STRUMENTI DI SVILUPPO NECESSARI E FAREMO LA CONOSCENZA DEI PRINCIPI DI BASE CHE REGOLANO IL FUNZIONAMENTO DEL SISTEMA MOBILE REALIZZATO DA GOOGLE

Meno di due anni fa Google ha rilasciato una versione preliminare del kit di sviluppo di Android, il suo nuovo SO dedicato agli smartphone. Futurologi e semplici appassionati si divisero immediatamente tra entusiasti e scettici. I detrattori, in particolar modo, hanno sempre visto in Android un esperimento, e non qualcosa di reale al quale i produttori di dispositivi avrebbero creduto. A loro favore ha deposto il fatto che, per un periodo piuttosto lungo, nessuno smartphone equipaggiato con Android ha fatto capolino sul mercato, benché il sistema e i suoi strumenti di sviluppo fossero ormai disponibili da parecchio tempo. La tecnica di Google, in realtà, era ed è ancora quella di sempre: far venire l'acquolina in bocca (e far parlare di sé) con versioni preliminari dei suoi software e dei suoi servizi.
Nel caso di Android, molti sviluppatori sono stati fidelizzati e fatti appassionare a un sistema che, allora, non era ancora sul mercato.
Nel frattempo le cose sono cambiate: Android è stato consolidato, e molti produttori di dispositivi mobili hanno aderito o stanno aderendo all'alleanza capeggiata da Google.
Grazie alle strategie di Google, esiste già una comunità molto ampia di sviluppatori, estremamente produttiva, che altri sistemi mobili non possono vantare. Migliaia di applicazioni sono state già sviluppate, e molte altre lo saranno nei prossimi tempi.
Il sistema appare inoltre stabile ed offre potenzialità molto ampie. Per questo motivo, a partire dal numero che state leggendo, ioProgrammo dedicherà ad Android un corso di programmazione a puntate.
Si comincia, naturalmente, con lo studio dell'architettura del sistema, l'installazione e l'utilizzo degli strumenti di sviluppo, un primo compendio sui principi di base della programmazione Android e lo sviluppo di una prima semplice applicazione del tipo "*Ciao Mondo*".

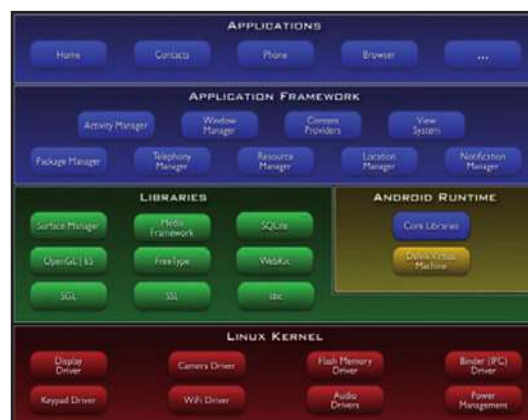



Fig. 1: L'architettura di Google Android

COME È FATTO ANDROID

Android, essendo un sistema operativo di moderna fattura, è abbastanza complesso. Anche se il suo target sono i dispositivi mobili, l'architettura di Android ha poco da invidiare a quelle dei comuni sistemi per desktop o laptop. Tale architettura è presentata schematicamente in Fig.1.
Come si evince dalla figura, Google ha attinto a piene mani dal mondo Open Source.
Il cuore di ogni sistema Android, tanto per cominciare, è un kernel Linux, versione 2.6.
Direttamente nel kernel sono inseriti i driver per il controllo dell'hardware del dispositivo: driver per la tastiera, lo schermo, il touchpad, il Wi-Fi, il Bluetooth, il controllo dell'audio e così via. Sopra il kernel poggiano le librerie fondamentali, anche queste tutte mutuate dal mondo Open Source. Da citare sono senz'altro OpenGL, per la grafica, SQLite, per la gestione dei dati, e WebKit, per la visualizzazione delle pagine Web. Insomma, nei prossimi mesi avremo di che discutere!
L'architettura prevede poi una macchina virtuale e una libreria fondamentale che, insie-

me, costituiscono la piattaforma di sviluppo per le applicazioni Android. Questa macchina virtuale si chiama *Dalvik*, e sostanzialmente è una Java Virtual Machine. Come verificheremo più tardi, alcune delle caratteristiche di Dalvik e della sua libreria non permettono di identificare immediatamente la piattaforma Java disponibile in Android con una di quelle di riferimento (Java SE, Java ME).
Nel penultimo strato dell'architettura è possibile rintracciare i gestori e le applicazioni di base del sistema. Ci sono gestori per le risorse, per le applicazioni installate, per le telefonate, il file system e altro ancora: tutti componenti di cui difficilmente si può fare a meno.
Infine, sullo strato più alto dell'architettura, poggiano gli applicativi destinati all'utente finale. Molti, naturalmente, sono già inclusi con l'installazione di base: il browser ed il player multimediale sono dei facili esempi.
A questo livello si inseriranno anche le applicazioni che, insieme, impareremo a sviluppare nell'arco di questo corso a puntate.

ANDROID SDK

Per sviluppare applicazioni in grado di girare su sistemi Android, è necessario installare sul proprio PC un apposito kit di sviluppo (SDK), che sia completo di emulatore, librerie e documentazione. Se avete già sviluppato per piattaforme quali Java ME o Windows Mobile, capite bene cosa intendiamo (ma se non l'avete mai fatto, non vi preoccupate: qui si spiega tutto). La prima buona notizia è che l'Android SDK è disponibile gratuitamente e senza discriminazioni per sistemi Windows, Linux e MacOS X. Come inizio, non c'è male.
È possibile scaricarlo collegandosi all'indirizzo: http://developer.android.com/sdk/
Vi verrà proposto di scaricare la più recente versione disponibile. Procedete pure al download del pacchetto adatto al vostro sistema.
Al momento della stesura di questo articolo, la versione scaricabile dalla pagina è la *1.5_r3* (che sta per 1.5 Release 3), ma se ne trovate di più recenti fate pure: non dovrebbero differire troppo da quella presa qui a riferimento.
L'installazione del kit è veramente semplice. L'unica cosa di cui bisogna accertarsi, prima di procedere, è di soddisfare i requisiti di base. In particolare, è richiesto che il sistema disponga già di un Java SDK (JDK) versione 5 o successiva. È strettamente indispensabile soddisfare questo requisito, poiché Android si programma in Java, e senza un JDK non è possibile compilare il codice. Dopo aver verificato i requisiti, è possibile procedere. Prendete l'archivio ZIP scaricato da Internet e scompattatelo dove meglio preferite. È tutto: l'Android SDK è già pronto all'uso! Al massimo si può perfezionare l'installazione aggiungendo alla variabile d'ambiente *PATH* del sistema operativo il percorso della cartella *tools* che è all'interno dell'Android SDK. Così facendo sarà più semplice invocare gli eseguibili del kit da riga di comando. L'operazione, ad ogni modo, non è indispensabile per un corretto funzionamento del kit. In questo corso, inoltre, cercheremo di servirci il meno possibile della riga di comando, anche se in alcune occasioni tornerà utile se non indispensabile.

ADT PER ECLIPSE

Benché Android SDK disponga di script che automatizzano l'installazione delle applicazioni, il lancio dell'emulatore e il debug del codice, lavorare in un ambiente integrato, con ogni opzione a portata di clic, è sicuramente più facile. Specie quando l'ambiente integrato si chiama Eclipse. Nel sito di Android contattato in precedenza è disponibile anche un plug-in per la celebre piattaforma di sviluppo Open Source. Questo add-on si chiama *Android Development Tools for Eclipse*, che abbreviato diventa ADT.
Il modulo, al momento della stesura di questo articolo, funziona con le più recenti versioni di

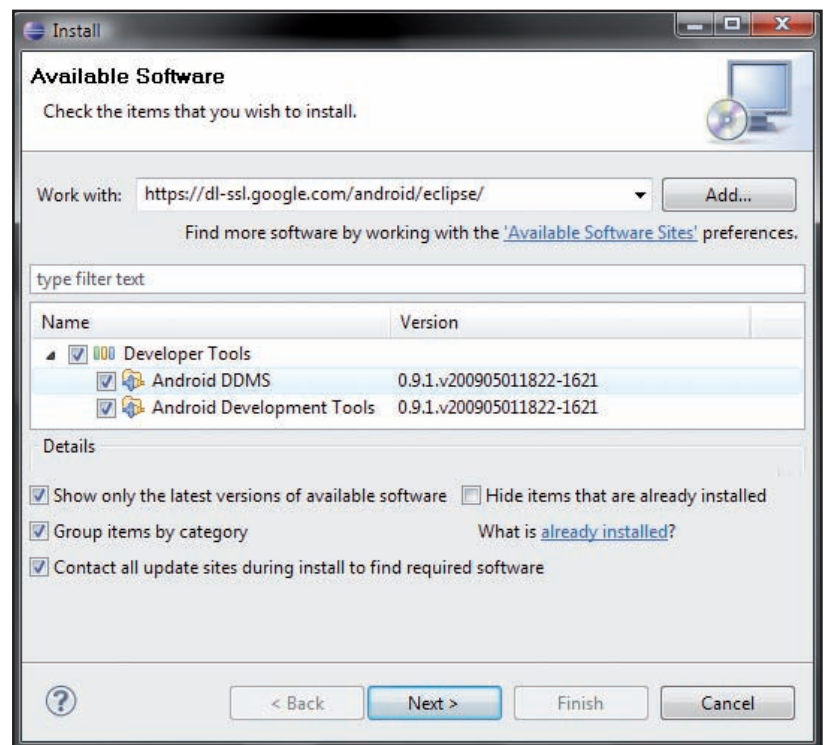



Fig. 2: Configurando la nuova fonte in Eclipse, è possibile scaricare ed installare automaticamente il plug-in per lo sviluppo del software Android (Eclipse 3.5)

Eclipse, che sono la 3.3, la 3.4 e anche la nuova 3.5. Può essere installato direttamente dall'interno della piattaforma di sviluppo.
Avviate Eclipse ed eseguite il wizard per l'installazione di nuovi componenti. In Eclipse 3.5 lo si fa con la voce di menu "*Help » Install New Software*". Nella 3.4 la voce di menu è "*Help » Software Updates*", e poi si deve selezionare la scheda "*Available software*". In Eclipse 3.3, infine, la voce di menu da richiamare è "*Help » Software Updates » Find and install*", scegliendo poi "*Search for new features to install*". Giunti a destinazione, scegliete l'opzione per aggiungere un nuovo sito remoto alla lista delle fonti presentate dal wizard (pulsante "*Add*"). L'indirizzo da specificare è:
https://dl-ssl.google.com/android/eclipse/r
A questo punto selezionate la voce corrispondente alla nuova fonte e procedete attraverso i singoli passi del wizard. Il plug-in per lo sviluppo del software Android sarà automaticamente scaricato e installato. Dopo il riavvio di Eclipse, recatevi immediatamente nella schermata delle preferenze dell'ambiente (voce di menu "Window » Preferences"). Qui troverete disponibile la nuova categoria "*Android*", nell'elenco sulla sinistra.
Selezionatela e impostate il percorso del vostro Android SDK: è necessario affinché Eclipse possa agganciare il kit di sviluppo. Durante questa fase dovreste anche ricevere un pop-up per l'accettadella licenza del plug-in.

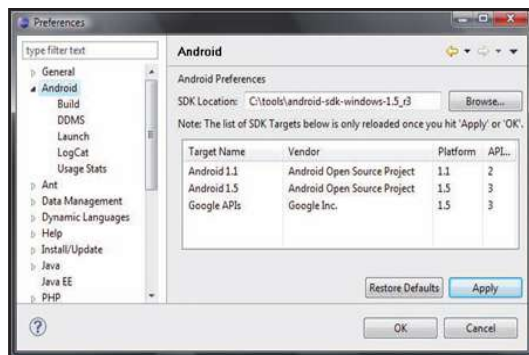



***Fig. 3:** Affinché il plug-in funzioni correttamente è necessario fornire il percorso dell'Android SDK*

Per ora è possibile tralasciare le altre possibili impostazioni collegate al plug-in: impareremo ad usarle più avanti, quando avremo preso più confidenza con l'ambiente.

GESTIONE DEGLI AVD

Il kit di sviluppo comprende un emulatore che ci consentirà di provare le nostre creazioni sul PC, prima di installarle su un reale dispositivo equipaggiato con Android. Per sviluppare le applicazioni, quindi, dobbiamo imparare a interagire con questo emulatore. Il primo concetto che si deve assimilare è quello che ha nome *Android Virtual Device* (AVD), cioè dispositivo *virtuale Android*. Nel nostro PC possiamo creare e configurare quanti dispositivi virtuali vogliamo. È come avere tanti differenti smartphone da utilizzare per i propri test, solo che invece di dispositivi di plastica e silicio si tratta di macchine virtuali, fatte cioè di puro software, da eseguire attraverso l'emulatore. In questo modo è anche possibile avviare contemporaneamente sullo stesso PC due o più dispositivi virtuali, ad esempio per testare un'applicazione che fa interagire più smartphone, come una chat o un gioco multiplayer. Impariamo allora a gestire l'elenco dei dispositivi virtuali configurati nel nostro Android SDK. È possibile gestirli da riga di comando, usando il comando *android* che è nella cartella *tools* del kit di sviluppo, ma l'uso dell'interfaccia di gestione resa disponibile da ADT in Eclipse ci renderà l'operazione senz'altro più agevole. Attivate la voce di menu *"Window » Android AVD Manager"*.

***Fig. 4:** La maschera di gestione dei dispositivi virtuali Android*

La finestra presenta la lista dei dispositivi virtuali configurati (inizialmente vuota) e la maschera per la creazione di una nuova istanza. Sfruttiamo questa possibilità e creiamo il nostro primo AVD. Le voci da compilare sono:

- **Name**: il nome che si vuole attribuire al dispositivo virtuale, ad esempio "*Android1*".
- **Target**: la tipologia del dispositivo. Sceglia mo Android 1.5. Creeremo così un disposi tivo virtuale compatibile con la versione 1.5 delle specifiche di Android.
- **SD Card**: qui è possibile dotare il dispositi vo virtuale di una scheda di memoria virtuale. È possibile specificare sia il percorso di un file di immagine di una scheda di me moria, se si vuole riutilizzare una memoria

virtuale esi stente, sia una dimensione di spazio, per creare una nuova memoria virtuale. Percorriamo quest'ultima strada e specifi chiamo il valore "64M". Verrà così creata una scheda di memoria virtuale di 64 MB.

- **Skin**: dall'elenco è possibile scegliere la riso luzione del dispositivo. Le scelte possibili sono HVGA-P (equivalente a 480x320), HV GA-L (320x480), QVGA-P (320x240) e QVGA-L (240x320). C'è poi una scelta di default chiamata semplicemente HVGA, che corri sponde comunque a HVGA-P. Lascia mola selezionata.

Dopo aver impostato questi valori, confermiamo l'operazione con il tasto "*Create AVD*". Il nuovo dispositivo virtuale entrerà a far parte dell'elenco gestito dal manager, e da ora potrà essere utilizzato per eseguire il debug e il test delle applicazioni.

PROVIAMO L'EMULATORE

Se non siete pratici nell'utilizzo di Android, prima di iniziare a programmare è meglio che ci prendiate confidenza. Esplorando le applicazioni di base potrete così entrare nell'ottica del sistema, per imparare i principi di funzionamento e di design delle sue applicazioni.
Potete avviare un dispositivo virtuale dall'esterno di Eclipse, al solo scopo di fare un "giro di ricognizione. Con il prompt dei comandi posizionatevi nella directory *tools* del kit di sviluppo. Lanciate ora un comando del tipo:
emulator @NomeAVD
A "*NomeAVD*" dovete sostituire il nome che, nel corso del paragrafo precedente, avete assegnato al dispositivo virtuale creato.
Ad esempio: *emulator @Android1*
Qualche istante di pazienza (al primo lancio anche qualcosa in più) e l'emulatore caricherà e renderà disponibile il dispositivo virtuale Android, in tutto il suo splendore. Con il mouse è possibile simulare il touchpad del dispositivo, cliccando sullo schermo. Fatevi un giro e prendete pure confidenza con l'ambiente. Come prima cosa divertitevi con le applicazioni di base, come il browser o la rubrica: vi aiuteranno molto nel comprendere i principi di utilizzo del sistema. Poi passate a del materiale più tecnico: il menu principale contiene la voce "*Dev Tools*", che raccoglie una serie di strumenti dedicati a chi Android vuole programmarlo, e non solo farci un giro di prova. Tra questi spicca l'emulatore di terminale, che permette di avere una shell di sistema per un'interazione di più basso livello con il dispositivo.

Fig. 5: Configurazione dell'emulatore di Android

CIAO, MONDO ANDROIDE!

È venuto il momento di utilizzare ADT e l'emulatore per programmare la nostra prima applicazione Android.
Naturalmente sarà una variante del classico "Ciao, Mondo!".
Avviate Eclipse. Grazie ad ADT disponete ora di una nuova categoria di progetto, chiamata "*Android Project*". Create un progetto di questo tipo.Nel wizard di creazione del progetto utilizzate la configurazione che viene riportata di seguito:

- **Project name**: *CiaoMondoAndroide*
- **Build target**: selezioniamo "Android 1.5".
- **Application name**: *Ciao Mondo*
- **Package name**: *it.ioprogrammo.helloandroid*
- **CreateActivity**: *CiaoMondoAndroideActivity*

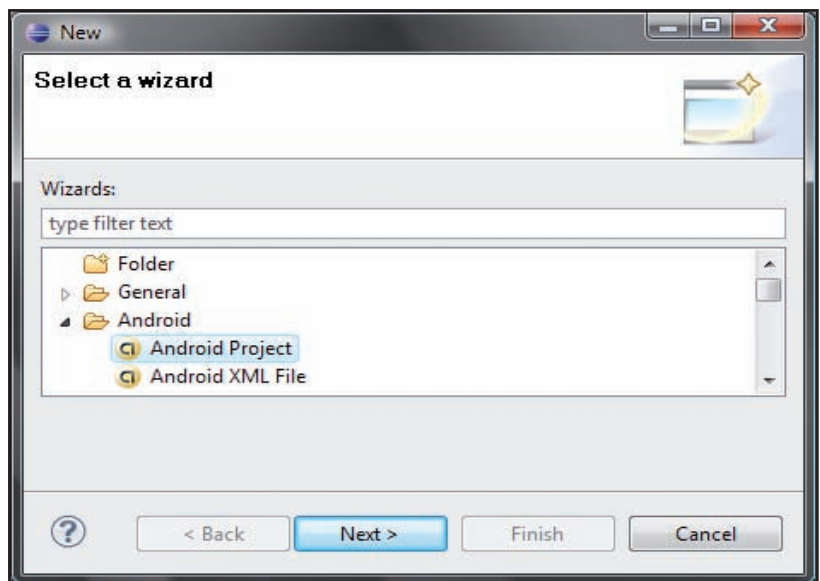



Fig. 6: Il nuovo tipo di progetto "Android Project" è ora disponibile in Eclipse

Il progetto, a questo punto, può essere creato, azionando il tasto "*Finish*".

Fig. 7: Wizard di creazione di un nuovo "Android Project" di Eclipse

Eclipse popolerà automaticamente il progetto, inserendo le librerie di Android e la struttura di base dei progetti per questa piattaforma. In uno slancio di generosità, Eclipse provvederà anche alla creazione della prima classe della soluzione, chiamata *CiaoMondoAndroideActivity* (come specificato alla voce "*Create Activity*") e inserita nel pacchetto *it.ioprogrammo.hello android* (come alla voce "*Package name*"). Aprite il codice della classe e modificatelo alla seguente maniera:

```
package it.ioprogrammo.helloandroid;
import android.app.Activity;
import android.os.Bundle;
import android.widget.TextView;
public class CiaoMondoAndroideActivity extends Activity
{
@Override public void onCreate(Bundle save
dInstanceState) {super.onCreate(savedInstanceState);
  TextView tv = new TextView(this);
  tv.setText("Ciao, Mondo Androide!");
  setContentView(tv);
}
}
```

Ora selezionate la radice del progetto "*CiaoMondoAndroide*", attivate il menu contestuale da tasto destro e lanciate la voce "*Run As » Android Application*".

L'emulatore verrà caricato. Eclipse provvederà

Fig. 8: Eclipse mette a disposizione l'opzione di avvio "Android Application"

automaticamente a installare al suo interno l'applicazione "*CiaoMondoAndroide*", per poi avviarla non appena l'operazione sarà completata. È fatta: il vostro primo software per Android sta girando davanti ai vostri occhi.

Successivamente, accedendo alle configurazioni di esecuzione (voce di menu "*Run » Run Configurations*" in Eclipse 3.5 e 3.4, "*Run » Open Run Dialog*" in Eclipse 3.3), sarà possibile alterare i parametri di avvio dell'emulatore e dell'applicazione. Tra questi, anche il dispositivo virtuale sul quale sarà installato e avviato il software.

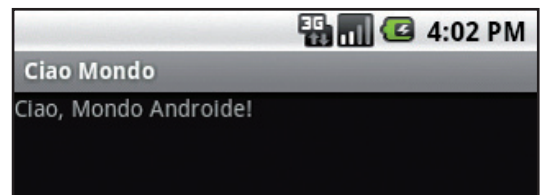



Fig. 9: L'applicazione "Ciao Mondo", eseguita dall'emulatore

Vi consigliamo di fare qualche esperimento. Provate, ad esempio, a creare differenti AVD, collaudando così il software con schermi di differenti dimensioni e proporzioni. Un altro esperimento interessante, che vi consiglio di compiere prima di procedere oltre, è l'utilizzo del debugger di Eclipse con l'applicazione Android. Ponete un breakpoint sulla classe realizzata e avviate di nuovo emulatore ed applicazione, questa volta in modalità debug.

DALVIK E LE LIBRERIE ANDROID

Superata la prova del primo progetto Android, torniamo ad occuparci dei concetti fondamentali per la programmazione in questo ambiente. Come abbiamo appreso e dimostrato, la piattaforma di sviluppo è di natura Java. Tuttavia si tratta di una piattaforma particolare e personalizzata, che vale la pena approfondire.

La macchina virtuale, chiamata *Dalvik*, sembra

NOTA

OPEN HANDSET ALLIANCE

Per essere precisi, dietro Android non c'è soltanto Google. Il colosso di Mountain View ha fatto la prima mossa e di sicuro è l'attore di maggior peso, tuttavia l'evoluzione di Android è curata da un consorzio denominato Open Handset Alliance. Del gruppo, oltre a Google, fanno parte numerosi altri nomi interessanti, tra cui HTC (la prima a produrre dispositivi equipaggiati con Android), Intel, Motorola, Samsung, LG e molti altri. C'è anche Telecom Italia. Per approfondire: http://www.openhandsetalliance.com/

essere una Java Virtual Machine, ma in realtà non lo è del tutto. Ci spieghiamo meglio: una Java Virtual Machine esegue del codice *bytecode*, giusto? Ecco, la Dalvik Virtual Machine non esegue bytecode standard, ma un altro linguaggio, chiamato *DEX* (*Dalvik EXecutable*), studiato appositamente per una migliore resa in uno smartphone. Con l'Android SDK ed Eclipse, ad ogni modo, ci sembrerà di utilizzare una regolare Java Virtual Machine. L'ambiente di sviluppo, infatti, provvede automaticamente alla generazione del codice DEX, ri-compilando il bytecode che a sua volta è frutto di una prima comune compilazione Java. Per noi sarà tutto trasparente. Questa peculiarità di Dalvik, quindi, non influenzerà il nostro modo di programmare. La stessa considerazione, invece, non può essere fatta riguardo la libreria di base che affianca Dalvik. Aprite il documento al percorso *docs/reference/packages.html,* nel vostro Android SDK. È l'indice dei package Java compresi nella libreria di base.
Scorretela velocemente e traete pure le prime conclusioni. C'è parecchio della Standard Edition di Java, ma non c'è tutto.
Ad esempio non ci sono AWT e Swing.
I pacchetti fondamentali, però, ci sono tutti, ed appaiono in larga misura identici a come li vuole Sun. Davvero poco viene dalla Micro Edition, praticamente nulla.
La piattaforma Java ME è stata snobbata da Android, che le ha preferito una libreria più simile a quella di un sistema desktop. Non passano poi inosservati i tanti package con prefisso *android* che, naturalmente, sono esclusivi di questa speciale piattaforma. Servono per l'interazione diretta con le funzionalità del sistema sottostante. Ad esempio: il package *android.widget* contiene i componenti custom di Android per la costruzione delle interfacce grafiche (in *CiaoMondoAndroide* abbiamo usato *TextView*); nel pacchetto *android.graphics* ci sono le funzioni primitive per la grafica di più basso livello; in *android.location* ci sono gli strumenti per interagire con un eventuale ricevitore GPS compreso nel dispositivo.
Ciascuno dei pacchetti Android, naturalmente, meriterebbe una trattazione estesa e completa, tanti sono i possibili campi di applicazione. Ne emerge il profilo di una piattaforma di sviluppo complessa, perché molto ricca, ma semplice, perché ordinata e perché condivide parecchio con l'edizione tradizionale di Java.
Il consiglio, naturalmente, è quello di tenere sempre a portata di mano la documentazione delle API di Android. Fatevi poi guidare dalla curiosità: date pure una prima occhiata alle classi che più stuzzicano la vostra fantasia.

PRINCIPI DI PROGRAMMAZIONE

Chi programma con Java ME sa che le *MIDlet* sono il mattone fondamentale delle applicazioni MIDP; chi crea applicazioni Web con Java EE non può ignorare cosa sia una *Servlet*; persino i programmatori meno esperti sanno che le applicazioni Java, per girare in un browser, devono essere inglobate in una *Applet*.
Tutto questo per dire che ciascun ambiente, Java e non, dispone dei suoi mattoni fondamentali, che lo sviluppatore può estendere e implementare per trovare un punto di aggancio con la piattaforma.
Android non sfugge alla regola, anzi la amplifica. A seconda di quel che si intende fare è disponibile un diverso modello. Android fornisce quattro mattoni di base:

- ***Attività***
  Le attività sono quei blocchi di un'applicazione che interagiscono con l'utente utilizzando lo schermo e i dispositivi di input messi a disposizione dallo smartphone.
  Comunemente fanno uso di componenti UI già pronti, come quelli presenti nel pacchetto *android.widget*, ma questa non è necessariamente la regola. La classe dimostrativa *CiaoMondoAndroideActivity* è un'attività. Le attività sono probabilmente il modello più diffuso in Android, e si realizzano estendendo la classe base *android.app.Activity*.

- ***Servizio***
  Un servizio gira in sottofondo e non interagisce direttamente con l'utente.
  Ad esempio può riprodurre un brano MP3, mentre l'utente utilizza delle attività per fare altro. Un servizio si realizza estendendo la classe *android.app.Service*.

- ***Broadcast Receiver***
  Un Broadcast Receiver viene utilizzato quando si intende intercettare un particolare evento, attraverso tutto il sistema. Ad esempio lo si può utilizzare se si desidera compiere un'azione quando si scatta una foto o quando parte la segnalazione di batteria scarica. La classe da estendere è *android. content.BroadcastReceiver*.

- ***Content Provider***
  I Content Provider sono utilizzati per esporre dati e informazioni. Costituiscono un canale di comunicazione tra le differenti applicazioni installate nel sistema. Si può creare un Content Provider estendendo la classe astratta *android.content.ContentProvider*.

NOTA

SITO DI RIFERIMENTO
Il principale sito Web di riferimento per tutti gli sviluppatori Android del mondo è, naturalmente, quello ufficiale, raggiungibile all'indirizzo: http://developer.android.com/

NOTA

DOCUMENTAZIONE
Tantissima documentazione (soltanto in inglese, però) è messa a disposizione nella cartella docs dell'Android SDK.

Un'applicazione Android è costituita da uno o più di questi elementi. Molto frequentemente, contiene almeno un'attività, ma non è detto che debba sempre essere così.

I PACCHETTI APK

Le applicazioni Android sono distribuite sotto forma di file *APK (Android Package)*. Al loro interno vengono raccolti gli eseguibili in formato DEX, le eventuali risorse associate e una serie di descrittori che delineano il contenuto del pacchetto. In particolare, nel cosiddetto *manifesto*, vengono dichiarate le attività, i servizi, i provider e i receiver compresi nel pacchetto, in modo che il sistema possa agganciarli e azionarli correttamente.
Torniamo, in Eclipse, sul progetto *CiaoMondoAndroide*. Al suo interno troverete un file chiamato *AndroidManifest.xml*, fatto come segue:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://
schemas.android.com/apk/res/android"
package="it.ioprogrammo.helloandroid"
android:versionCode="1"
android:versionName="1.0">

<application android:icon="@drawable/icon" an
droid:label="@string/app_name">
<activity android:name=".CiaoMondoAndroi
deActivity"
android:label="@string/app_name">
<intent-filter>
<action android:name="android.intent.
action.MAIN" />
<category android:name="
android.intent.category.LAUNCHER" />
</intent-filter>
</activity>
</application>
<uses-sdk android:minSdkVersion="3" />
</manifest>
```

È il manifesto descrittore citato poco fa. Al suo interno potete e dovete dichiarare i componenti del vostro software. Eclipse, all'atto di creazione del progetto, ha già eseguito su di esso alcune configurazioni iniziali.
Ad esempio ha registrato l'attività *CiaoMondoAndroideActivity*, ha specificato le proprietà generali dell'applicazione e ha anche generato e impostato un'icona per il nostro programma (*res/drawable /icon.png*). Ovviamente queste scelte possono essere alterate, e nuovi componenti possono essere aggiunti al progetto.
Con lo speciale editor visuale messo a disposizione da Eclipse, vi risulterà tutto molto semplice: è sufficiente fare un po' di pratica e approfondire di volta in volta l'aspetto d'interesse.
Una volta che il lavoro è stato completato, è possibile esportare il file APK da distribuire ai fortunati possessori di un sistema Android.
Prima di distribuire in giro il pacchetto è però necessario apporre su di esso una firma digitale. In caso contrario, Android non potrà installarne i contenuti.
Questo è l'unico vincolo imposto dal sistema.
Il fatto che un pacchetto debba essere firmato non deve preoccupare lo sviluppatore: non è necessario che una certification authority riconosca la chiave utilizzata per la firma.
Di conseguenza è possibile firmare un pacchetto APK anche servendosi di un certificato "fatto in casa". In parole semplici: non bisogna pagare nessuno perché i nostri software siano autorizzati, possiamo fare tutto da noi.
In Eclipse, ancora una volta, è questione di un clic: aprite il menu contestuale sulla radice del progetto (tasto destro del mouse, in Windows) e selezionate la voce "*Android Tools » Export Signed Application Package*".
Al secondo step del wizard di generazione del pacchetto, vi verrà chiesto da dove prelevare la firma digitale.
Solitamente gli oggetti di questo tipo vengono raccolti e conservati all'interno di un keystore. In un keystore, cioè, ci sono più firme digitali. Se non avete mai formato un keystore in precedenza, o se semplicemente ne volete iniziare uno nuovo, selezionate l'opzione "*Create new keystore*".
Il keystore verrà conservato all'interno di un file, il cui percorso va obbligatoriamente spe-

NOTA

IL TOOL ADB

Oltre all'emulatore, la cartella *tools* dell'Android SDK contiene un altro strumento molto interessante, chiamato *adb*. Si utilizza da riga di comando. Lanciatelo senza parametri, e avrete un veloce aiuto in linea sulle funzionalità messe a disposizione. In particolare, mentre l'emulatore è in esecuzione, i comandi *adb install* e *adb uninstall* possono essere utilizzati per installare e rimuovere applicazioni dal dispositivo, mentre lo speciale comando adb shell permette di aprire una shell sul sistema Android emulato.

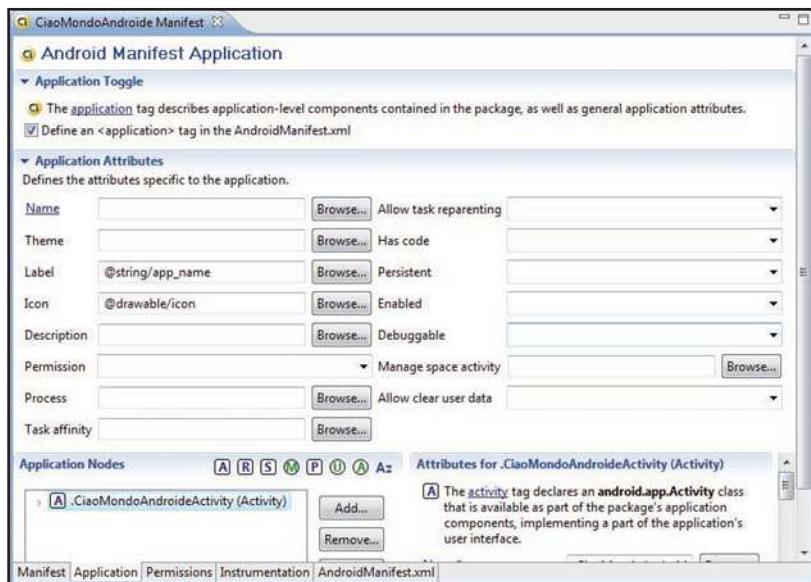



Fig. 10: Lo speciale editor messo a disposizione da Eclipse per il file AndroidManifest.xml

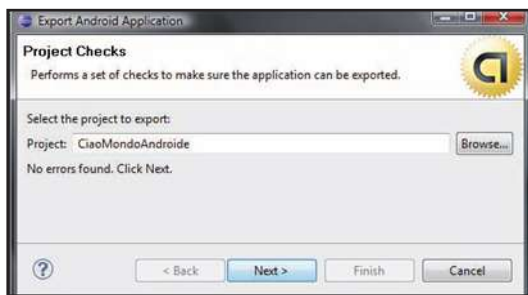



Fig. 11: La prima schermata di generazione di un pacchetto APK firmato a partire dal progetto CiaoMondoAndroide

cificato. Scegliete dove riporre il keystore (nel caso in Fig.13, la directory è C:*keystores*) e date un nome a vostro piacimento a file (*android_keystore*, nel caso in immagine). Non c'è bisogno di usare un'estensione particolare per il nome del file. È invece buona pratica proteggere i propri keystore con una password, in modo che le nostre firme digitali

Fig. 12: La creazione di un nuovo keystore

non possano essere utilizzate nel caso in cui qualcuno ci rubi il file. Pertanto abbiate cura di impostare una password sufficientemente sicura.
Visto che il keystore appena creato è vuoto, il passo successivo del wizard ci fa creare una chiave, cioè una firma digitale. Dobbiamo inserire il nome della chiave (detto *alias*), la password per l'utilizzo della chiave, una validità in anni (di solito si usa il valore *25*) e i dati anagrafici di base del firmatario (nome e cognome).
Superata la fase di creazione o selezione del keystore e della chiave, il wizard fa scegliere dove salvare il pacchetto APK che sarà generato. Scegliete la destinazione e concludete l'operazione.
È fatta: il pacchetto è stato generato e firmato. Potete ora installarlo su un dispositivo Android reale, in plastica, metallo e silicio.

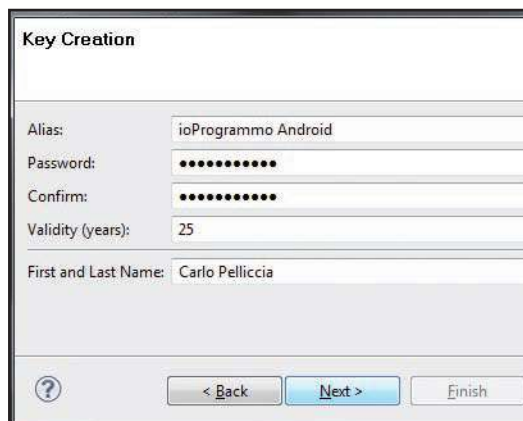



Fig. 13: La maschera per la creazione di un nuova firma digitale

COSA TRATTEREMO NEI PROSSIMI ARTICOLI DEL CORSO

Nel prossimo numero della rivista andremo un po' più al di dentro della faccenda, approfondendo i principi di programmazione di Google Android, conoscendo più intimamente la struttura delle sue applicazioni e iniziando la rassegna delle API messe a disposizione dal sistema operativo. Per tutto l'arco di questo corso si cercherà sempre di rimanere "con i piedi per terra", affrontando i differenti argomenti in maniera sistematica e concludendo sempre le lezioni con un esempio pratico e funzionale, in modo da fornire ai lettori un'immediata implementazione dei concetti appresi.

Carlo Pelliccia

Carlo Pelliccia lavora presso 4IT (www.4it.it), dove si occupa di analisi e sviluppo software per piattaforme Java. Nella sua carriera di technical writer ha pubblicato cinque manuali ed oltre centocinquanta articoli, molti dei quali proprio tra le pagine di ioProgrammo. Il suo sito, che ospita anche diversi progetti Java Open Source, è disponibile all'indirizzo www.sauronsoftware.it

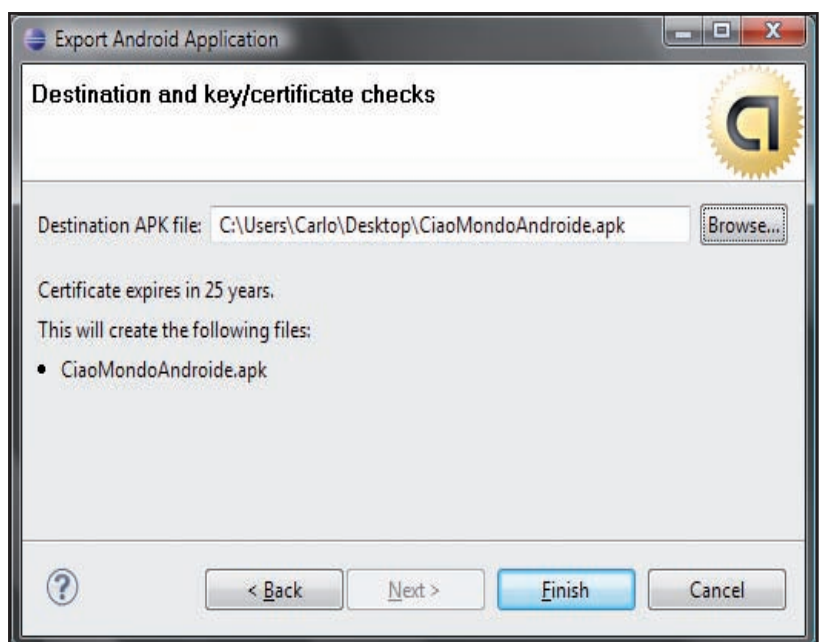



Fig. 14: La selezione del file su cui sarà salvato il pacchetto APK

LE RISORSE ESTERNE IN GOOGLE ANDROID

IN QUESTO SECONDO ARTICOLO IMPAREREMO A MANIPOLARE LE RISORSE ESTERNE. SCOPRIREMO CHE ANDROID RENDE FACILE L'UTILIZZO DI TUTTO QUELLO CHE, PUR NON ESSENDO CODICE, È INDISPENSABILE AL CORRETTO FUNZIONAMENTO DI UN'APPLICAZIONE

Se c'è un aspetto di Android dal quale si evince la modernità di questo sistema, è la sua maniera di gestire le risorse e i dati. Nelle piattaforme di sviluppo meno moderne, spesso e volentieri, le risorse esterne come i dati di configurazione, i messaggi di interfaccia, le immagini o altro materiale simile, sono trattate senza alcun riguardo speciale. Android, invece, richiede che i progetti siano organizzati in una certa maniera. La corretta gestione delle risorse in questa piattaforma, è importante tanto quanto la stesura del codice. In un certo senso, con Android non si può imparare a programmare se prima non si apprende come organizzare e richiamare le risorse. Perciò, dedichiamo all'argomento questo secondo articolo del corso.

Fig. 1: La struttura, in Eclipse, di un progetto Android appena creato

LA STRUTTURA DEI PROGETTI ANDROID

Avviamo Eclipse e torniamo sul progetto *Ciao MondoAndroide*, realizzato il mese scorso per dimostrare le funzionalità di base del kit di sviluppo per Android. Quando abbiamo creato il progetto, Eclipse ha predisposto per noi un albero di cartelle, all'interno del quale sono stati generati automaticamente diversi file. Guardate la Fig.1, che mostra la situazione del file system all'atto di creazione del progetto.

Tra i file generati automaticamente c'è *AndroidManifest.xml*, cioè il descrittore dell'applicazione, che già abbiamo iniziato a conoscere. Torneremo ad approfondirlo mano a mano che gli argomenti trattati ce ne daranno occasione. Oltre al descrittore c'è il file *default.properties*, poco rilevante per noi, poiché serve esclusivamente al sistema di build automatico. Ci sono poi delle directory: *src*, *assets*, *res* e *gen*. La prima, *src*, è quella dove dobbiamo andare a realizzare i package e le classi della nostra applicazione. Le cartelle *res* e *assets* servono per ospitare le risorse esterne necessarie all'applicazione, come le immagini, i file audio e altro ancora. La cartella *res*, in particolar modo, gode di una speciale struttura predefinita, formata dalle tre sotto-directory *drawable*, *layout* e *values*. La prima, *drawable*, serve per le immagini utilizzate dal software, mentre *layout* e *values* ospitano dei speciali file XML utili per definire in maniera dichiarativa l'aspetto dell'applicazione e i valori utilizzati al suo interno. Oltre a *src*, *assets* e *res* c'è infine la cartella *gen*, che contiene la speciale classe chiamata *R*, probabile abbreviazione di *Resources*. Invocando questa classe, infatti, è possibile richiamare via codice le risorse memorizzate sotto la directory *res*. Impareremo oggi stesso come farlo. Sappiate comunque che la classe *R* viene generata automaticamente dal sistema e non deve mai essere modificata a mano.

GESTIONE DEI VALORI

Il primo tipo di risorse che impareremo a manipolare sono i valori. Si tratta di coppie chiave-

valore dichiarate all'interno dei file XML che sono al percorso di progetto *res/values*. Eclipse, per default, crea a questo percorso il file *strings.xml*, pensato per raccogliere le stringhe usate dall'applicazione che sarà sviluppata.
Ad ogni modo potete rinominare il file o aggiungerne quanti altri ne volete, al fine di categorizzare al meglio i valori necessari alla vostra applicazione. L'importante è che tutti i file presenti nella cartella *values* seguano il seguente modello:

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
  <!-- valori qui -->
</resources>
```

All'interno del tag *<resources>* … *</resources>* è possibile dichiarare differenti tipi di valori. Supponiamo di voler dichiarare un valore di tipo stringa chiamato *nome* e con contenuto *Carlo*:

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
            <string name="nome">Carlo</string>
</resources>
```

Ci sono numerosi tipi di dati supportati. Ecco un elenco completo:

- **Stringhe**, con il tag *<string>*.
- **Colori**, con il tag *<color>* e con valori espressi in forma esadecimale secondo i modelli *#RRGGBB* o *#AARRGGBB* (*AA* sta per il canale *alpha*, che regola la trasparenza del colore). Ad esempio: *<color name="rosso">#FF0000</color>*
- **Misure e dimensioni**, con il tag *<dimen>* e con valori numerici decimali accompagnati da un'unità di misura che può essere *px* (pixel), *in* (pollici), *mm* (millimetri), *pt* (punti a risoluzione 72dpi), *dp* (pixel indipendenti dalla densità) e *sp* (pixel indipendenti dalla scala). Ad esempio: *<dimen name="lato">180px</dimen>*
- **Rettangoli di colore**, con il tag *<drawable>*. I valori possibili sono colori esadecimali come nel caso del tag *<color>*. Ad esempio: *<drawable name="verde">#00FF00</drawable>*
- **Array di interi**, con il tag *<integer-array>*. Gli elementi dell'array vanno espressi con più occorrenze del tag annidato *<item>*. Ad esempio:
  <integer-array name="numeriPrimi">
  <item>2</item><item>3</item>
  <item>5</item><item>7</item>
  </integer-array>
- **Array di stringhe**, con il tag *<string-array>*. Anche in questo caso si usa il tag *<item>*. Ad esempio:
  <string-array name="nomi">
  <item>Carlo</item>
  <item>Claudia</item>
  <item>Nami</item>
  </string-array>
- **Stili e temi**, con il tag *<style>*. Servono per creare degli stili di disegno ed impaginazione, un po' come fanno i fogli di stile CSS nel caso di HTML. Dentro il tag *<style>* vanno inseriti dei tag *<item>* con le singole voci che compongono lo stile. Gli stili possono essere applicati alle interfacce grafiche. Di certo ne riparleremo più avanti. Eccovi comunque un esempio:
  <style name="titolo">
  <item name="android:textSize">18sp</item>
  <item name="android:textColor">#000088</item>
  </style>

Usando Eclipse, comunque, non c'è bisogno di imparare l'elenco a memoria: qualsiasi file XML posto sotto la directory *res/values* viene automaticamente lavorato con un apposito editor.

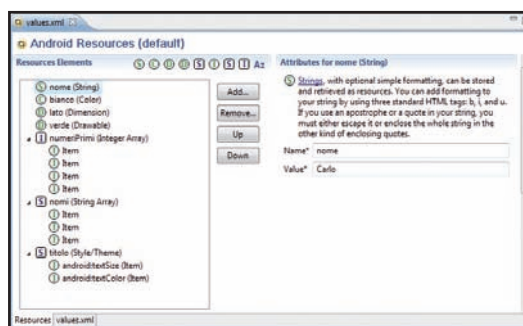



***Fig. 2:** L'editor di Eclipse gestisce i file XML*

RICHIAMARE LE RISORSE DA XML

Come scritto in apertura, la modernità di Android può essere evinta proprio dalla sua maniera di gestire le risorse. Le piattaforme di una volta non concedevano sistemi agevolati, finendo così per favorire l'accoppiamento fra codice e dati. Tuttora non è raro vedere dei sorgenti in Java, in C o in qualsiasi altro linguaggio, con valori e messaggi digitati direttamente dentro il codice. Questa pratica non è corretta ed è sconsigliata da ogni manuale: è sempre meglio separare i dati dal codice, perché in questa maniera il software è più facile sia da realizzare sia da mantenere. Android intende favorire la pratica del disaccoppiamento fra dati e codice, e lo fa attraverso gli strumenti che stiamo prendendo in considerazione oggi. I valori dichiarati nei file XML sotto *values*, così come tutte le altre risorse della cartella *res* e delle sue annidate, sono trattati dal sistema in maniera speciale.
Il kit di sviluppo, infatti, fornisce delle agevola-

DIFFERENZA TRA RES E ASSETS

La differenza tra le cartelle *res* e ***assets*** è poco evidente, eppure c'è. La directory *res* è pensata per gestire le risorse in maniera struttura, ed infatti è suddivisa in sottocartelle. Tutte le risorse posizionate in res vengono prese in esame dal sistema di build e riferite nella speciale classe *R*. Quelle dentro *res*, dunque, sono delle risorse *gestite*. Sotto *assets*, invece, è possibile depositare qualsiasi file si desideri senza che il sistema di build esegua un'analisi preventiva e crei il riferimento in *R*.
Le risorse esterne conservate nella directory *assets* possono essere caricate servendosi della classe *android.content.res.AssetManager*. Nella maggior parte dei casi, comunque, non c'è bisogno di ricorrere alla cartella assets, poiché res offre una maniera semplificata e completa per l'accesso alle risorse.

zioni per richiamare le risorse dalle varie parti del software. Sostanzialmente un'applicazione Android è costituita da file dichiarativi XML e da classi Java. Sia in un caso sia nell'altro, ci sono scorciatoie per richiamare le risorse incluse in *res*. Cominciamo dal caso XML e prendiamo a riferimento il più importante dei file di questo tipo: *AndroidManifest.xml*. Quando, al suo interno, si dichiarano i dettagli dell'applicazione, è possibile scrivere qualcosa come:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android=
        "http://schemas.android.com/apk/res/android"
    package="mypackage"
    android:versionCode="1"
    android:versionName="1.0">
  <application android:label="LaMiaApplicazione">
    ...
  </application>
</manifest>
```

Il nome dell'applicazione, cioè *LaMiaApplicazione*, è stato in questo caso digitato direttamente dentro il codice XML. Con Android questo è corretto, tuttavia si può fare di meglio. Si può includere il titolo dell'applicazione nel file *res/values/strings.xml*, alla seguente maniera:

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
  <string
    name="app_name">LaMiaApplicazione</string>
</resources>
```

A questo punto il descrittore dell'applicazione può essere riscritto come segue:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android=
        "http://schemas.android.com/apk/res/android"
    package="mypackage"
    android:versionCode="1"
    android:versionName="1.0">
  <application android:label="@string/app_name">
  </application>
</manifest>
```

Anziché scrivere "*LaMiaApplicazione*", si è usato il riferimento *@string/app_name*. Questa scorciatoia, come intuibile, viene sostituita dalla risorsa di tipo stringa con nome *app_name*, che nel file *strings.xml* abbiamo dichiarato essere proprio "*LaMiaApplicazione*".

La regola generale per richiamare una risorsa in un file XML, quindi, è basata sul modello: *@tipo/nome*. I tipi validi sono:

- ***@array***, per gli array.
- ***@color***, per i colori.
- ***@dimen***, per le dimensioni.
- ***@drawable***, per i valori *drawable*, ma anche per le immagini messe in *res/drawable*.
- ***@layout***, per richiamare i layout presenti nella cartella *res/layout*.
- ***@raw***, per i file nella cartella *res/raw* (cfr. box laterale).
- ***@string***, per le stringhe.
- ***@style***, per gli stili.

Con *@drawable*, in particolar modo, è possibile riferire sia i valori dichiarati con i tag *<drawable>* in *res/values*, sia le immagini conservate nella cartella *res/drawable*. Ad esempio, se in *res/drawable* viene messa un'icona chiamata *icon.png*, sarà possibile richiamarla con la formula *@drawable/icon*. Ad esempio lo si può fare in *AndroidManifest.xml*, per associare l'icona all'applicazione:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android=
        "http://schemas.android.com/apk/res/android"
    package="mypackage"
    android:versionCode="1"
    android:versionName="1.0">
  <application
        android:label="@string/app_name"
        android:icon="@drawable/icon">
        ...
  </application>
</manifest>
```

RICHIAMARE LE RISORSE DA JAVA

Valori e risorse possono essere richiamati da codice Java servendosi della classe *android.content.res.Resources*. Stando all'interno di una attività, cioè di una classe che estende *android.app.Activity*, è sufficiente richiamare il metodo *getResources()* per ottenere il punto d'accesso alle risorse dell'applicazione: *Resources res = getResources();* Una volta ottenuto l'oggetto, è possibile invocare su di esso la seguente serie di metodi:

- ***public int getColor(int id)***
  Restituisce il colore avente l'identificativo di risorsa specificato.
- ***public float getDimension(int id)***
  Restituisce la dimensione avente l'identificativo di risorsa specificato.
- ***public Drawable getDrawable(int id)***
  Restituisce l'oggetto disegnabile avente l'identificativo di risorsa specificato.

NOTA

ANIM, RAW E XML

Oltre a *drawable*, *layout* e *values*, che Eclipse introduce automaticamente in ogni nuovo progetto, la cartella *res* può ospitare anche le sotto-directory *anim*, *raw* e *xml*. In *anim* si inseriscono le animazioni, che possono essere programmate in maniera dichiarativa con uno speciale formato XML; sotto *raw* è possibile inserire qualsiasi tipo file, ad esempio un audio da riprodurre all'interno dell'applicazione; in *xml* un qualsiasi file XML, che Android provvederà automaticamente a decodificare. Ne riparleremo nei prossimi mesi.

NOTA

DISTINGUERE I FILE CON I VALORI

Benché, teoricamente, sia possibile mettere in *res/values* un solo file XML con ogni tipo di valore possibile, le linee guida dello sviluppo Android consigliano di distinguere i file per topic, cioè per argomento. Solitamente si consiglia di raccogliere le stringhe in un file *strings.xml*, i colori (e i drawable) in *colors.xml*, le dimensioni in *dimens.xml* e gli stili in *styles.xml*.

- ***public int[] getIntArray(int id)***
  Restituisce l'array di interi avente l'identificativo di risorsa specificato.
- ***public String getString(int id)***
  Restituisce la stringa avente l'identificativo di risorsa specificato.
- ***public String[] getStringArray(int id)***
  Restituisce l'array di stringhe avente l'identificativo di risorsa specificato.

Tutti questi metodi agganciano la risorsa desiderata attraverso un identificativo numerico (*int id*). Ma come fare a conoscere gli ID associati alle risorse e ai valori inseriti nella cartella *res*? Semplice: attraverso la speciale classe autogenerata R! Al suo interno sono contenute delle sottoclassi statiche, una per ciascuna tipologia di risorsa presente nel progetto: *R.string*, *R.drawable*, *R.color* e così via. In ciascuno di questi gruppi vengono introdotti gli ID numerici che corrispondono alle risorse e ai valori conservati in res e nelle sue sotto-cartelle. Proviamo con un esempio pratico: facciamo il caso che nel file *res/values/strings.xml* sia stata dichiarata la stringa *app_name*. Per richiamarla da codice Java, stando all'interno di una attività, si deve fare alla seguente maniera:

```
Resources res = getResources();
String appName = res.getString(R.string.app_name);
```

CIAO MONDO RELOADED

Mettiamo a frutto le nozioni acquisite quest'oggi, assemblando per la seconda volta un esempio del tipo "Ciao, Mondo!". Questa volta, però, useremo la più corretta pratica delle risorse esterne per il nome dell'applicazione e per il messaggio presentato sullo schermo. Create il progetto Android in Eclipse e chiamatelo *TestResources*.
Il package di riferimento è *it.ioprogrammo.testresources*, mentre l'attività principale deve essere *TestResourcesActivity*. Non appena il progetto è pronto, andiamo ad editare il file *res/values/strings.xml*:

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
  <string name="app_name">TestResources</string>
  <string name="message">Ciao, Mondo Androide!</string>
</resources>
```

Sono state dichiarate due stringhe: *app_name* con valore "*TestResources*" e *message* con valore "*Ciao, Mondo Androide!*".
Andiamo ora su *AndroidManifest.xml*:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android=
        "http://schemas.android.com/apk/res/android"
    package="it.ioprogrammo.testresources"
    android:versionCode="1"
    android:versionName="1.0">
   <application android:icon="@drawable/icon"
                    android:label="@string/app_name">
     <activity android:name=".TestResourcesActivity"
                android:label="@string/app_name">
        <intent-filter>
           <action
     android:name="android.intent.action.MAIN" />
           <category android:name=
              "android.intent.category.LAUNCHER" />
        </intent-filter>
     </activity>
   </application>
   <uses-sdk android:minSdkVersion="3" />
</manifest>
```

Il nome dell'applicazione, citato ben due volte all'interno del file, è stato richiamato mediante il riferimento *@string/app_name*. Allo stesso modo, l'icona per l'applicazione, creata automaticamente da Eclipse al percorso *res/drawable/icon.png*, è stata riferita attraverso la dicitura *@drawable/icon*. Adesso tocca alla classe *TestResourcesActivity*, il cui codice è riportato di seguito:

```
package it.ioprogrammo.testresources;
import android.app.Activity;
import android.content.res.Resources;
import android.os.Bundle;
import android.widget.TextView;
public class TestResourcesActivity extends Activity {
@Override
public void onCreate(Bundle savedInstanceState) {
                    super.onCreate(savedInstanceState);
 Resources res = getResources();
 String message = res.getString(R.string.message);
 TextView tv = new TextView(this);
 tv.setText(message);
 setContentView(tv);
}
}
```

Il messaggio "*Ciao, Mondo Androide!*", questa volta, è stato caricato attraverso la classe *Resources* ed il riferimento *R.string.message*.

PROSSIMAMENTE

Nel prossimo appuntamento approfondiremo i principi di programmazione delle *attività*, uno dei mattoni fondamentali delle applicazioni Android.

Carlo Pelliccia

INTERNAZIONALIZZAZIONE

Se si punta ad un mercato internazionale, è bene che le applicazioni realizzate siano tradotte in più lingue. Android aiuta gli sviluppatori consentendo l'internazionalizzazione delle risorse. Supponiamo di voler realizzare un'applicazione sia in inglese sia in italiano. Prepariamo due differenti file *strings.xml* con tutti i messaggi di interfaccia, uno in inglese e l'altro in italiano. Adesso, invece della cartella values, creiamo le due cartelle *values-it* (per l'italiano) e *values-en* (per l'inglese) ed inseriamo al loro interno i due file. È fatta! I dispositivi Android che eseguiranno l'applicazione sceglieranno automaticamente quale file caricare, in base alla loro lingua predefinita.

L'AUTORE

Carlo Pelliccia lavora presso 4IT (www.4it.it), dove si occupa di analisi e sviluppo software per piattaforme Java. Nella sua carriera di technical writer ha pubblicato cinque manuali ed oltre centocinquanta articoli, molti dei quali proprio tra le pagine di ioProgrammo. Il suo sito, che ospita anche diversi progetti Java Open Source, è disponibile all'indirizzo www.sauronsoftware.it

COME IL SISTEMA GESTISCE LE ATTIVITÀ

TERZO APPUNTAMENTO. LE "ATTIVITÀ" SONO IL COMPONENTE SOFTWARE PIÙ UTILIZZATO DAI PROGRAMMATORI ANDROID. IN QUESTO ARTICOLO IMPAREREMO COS'È UN'ATTIVITÀ, COME VIENE GESTITA DAL SISTEMA E COME POSSIAMO REALIZZARNE DI NOSTRE

Le applicazioni Android, come si è accennato durante la prima lezione di questo corso, si compongono di quattro mattoni fondamentali: le attività (*activity*), i servizi (*service*), i *broadcast receiver* e i *content provider*. Ogni applicazione è formata da uno o più di questi mattoni. Non è detto che li contenga tutti: ad esempio potrebbe essere costituita da due attività e da un servizio, senza avere broadcast receiver né content provider. Nella stragrande maggioranza dei casi, comunque, le applicazioni comprendono almeno un'attività. Le attività, di conseguenza, sono il più fondamentale dei componenti di base delle applicazioni Android.

COS'È UN'ATTIVITÀ

Stando alla documentazione ufficiale, un'attività è "una singola e precisa cosa che l'utente può fare". Proviamo a indagare le implicazioni di questa affermazione. Partiamo dal fatto che l'utente, per fare qualcosa, deve interagire con il dispositivo.
Domandiamoci come avvenga, nel caso di uno smartphone, l'interazione tra l'uomo e la macchina. Un ruolo essenziale, naturalmente, è svolto dai meccanismi di input, come la tastiera e il touchscreen, che permettono all'utente di specificare il proprio volere. Le periferiche di input, tuttavia, da sole non bastano. Affinché l'utente sappia cosa può fare e come debba farlo, ma anche affinché il software possa mostrare all'utente il risultato elaborato, è necessario un canale aggiuntivo. Nella maggior parte dei casi questo canale è il display.
Nella superficie dello schermo il software disegna tutti quegli oggetti con cui l'utente può interagire (bottoni, menu, campi di testo), e sempre sullo schermo viene presentato il risultato dell'elaborazione richiesta. Il ragionamento ci porta alla conclusione che, per fare qualcosa con il dispositivo, è necessario usare lo schermo. Esiste perciò un parallelo tra il concetto di attività, in Android, e quello di finestra, in un sistema desktop, benché non siano esattamente la stessa cosa. In generale, ad ogni modo, possiamo assumere con tranquillità che le attività sono quei componenti di un'applicazione Android che fanno uso del display e che interagiscono con l'utente.
Dal punto di vista del programmatore, poi, possiamo spingerci oltre e semplificare ulteriormente.
In maniera più pragmatica, un'attività è una classe che estende *android.app.Activity*. L'autore del codice, realizzando l'attività, si serve dei metodi ereditati da *Activity* per controllare cosa appare nel display, per assorbire gli input dell'utente, per intercettare i cambi di stato e per interagire con il sistema sottostante.

CICLO DI VITA DI UN'ATTIVITÀ

In un sistema desktop il monitor è sufficientemente spazioso da poter mostrare più finestre simultaneamente. Perciò non è affatto raro lavorare con più programmi contemporaneamente attivi, le cui finestre vengono affiancate o sovrapposte. Gli smartphone, invece, funzionano diversamente. Prima di tutto il display è piccolo, e pertanto ha poco senso affiancare due o più finestre di applicazioni differenti. Poi non bisogna dimenticare che le risorse di calcolo sono modeste, e perciò non è buona cosa tenere simultaneamente in vita troppi programmi. Per questi motivi le attività di Android hanno carattere di esclusività. È possibile mandare in esecuzione più attività simultaneamente, ma soltanto un'attività alla volta può occupare il display. L'attività che occupa il display è in esecuzione e interagisce direttamente con l'utente.
Le altre, invece, sono ibernate e tenute nascoste in sottofondo, in modo da ridurre al minimo il consumo delle risorse di calcolo. L'utente, naturalmente, può ripristinare un'attività ibernata e riprenderla da dove l'aveva interrotta, riportandola in primo piano. L'attività dalla quale si sta allontanando, invece, sarà ibernata e mandata in sottofondo al posto di quella ripristinata. Il cambio di attività può anche avvenire a causa di un evento esterno. Il caso più ricorrente è quello della telefonata in arrivo: se

il telefono squilla mentre si sta usando la calcolatrice, quest’ultima sarà automaticamente ibernata e mandata in sottofondo. L’utente, conclusa la chiamata, potrà richiamare l’attività interrotta e riportarla in vita, riprendendo i calcoli esattamente da dove li aveva interrotti.
Visto che le attività ibernate, in termini di risorse di calcolo, non consumano nulla, in Android il concetto di chiusura delle attività è secondario e tenuto nascosto all’utente. Ciò, di solito, spiazza chi è al suo primo confronto con la programmazione dei dispositivi portatili. Le attività di Android non dispongono di un bottone “x”, o di un tasto equivalente, con il quale è possibile terminarle. L’utente, di conseguenza, non può chiudere un’attività, ma può solo mandarla in sottofondo. Questo, comunque, non significa che le attività non muoiano mai, anzi! Per prima cosa le attività possono morire spontaneamente, perché hanno terminato i loro compiti. Insomma, anche se il sistema non ci fornisce automaticamente un bottone “chiudi”, possiamo sempre includerlo noi nelle nostre applicazioni. In alternativa, la distruzione delle attività è completamente demandata al sistema. I casi in cui un’attività può terminare sono due:

- L’attività è ibernata e il sistema, arbitrariamente, decide che non è più utile e perciò la distrugge.
- Il sistema è a corto di memoria, e per recuperare spazio inizia a “uccidere” bruscamente le attività in sottofondo.

Esistono poi dei task manager di terze parti che permettono di killare le attività in sottofondo, ma non sono previsti nel sistema di base.
I differenti passaggi di stato di un’attività attraversano alcuni metodi della classe *Activity* che, come programmatori, possiamo ridefinire per intercettare gli eventi di nostro interesse.
La Fig.1 illustra la sequenza di chiamate ai metodi di Activity eseguite durante i passaggi di stato dell’attività. Entriamo nel dettaglio:

Fig. 1: Ciclo di vita di un’attività. Sono illustrate le chiamate ai metodi che è possibile ridefinire per intercettare i passaggi di stato

- ***protected void onCreate(android.os.Bundle savedInstanceState)***
  Richiamato non appena l’attività viene creata. L’argomento savedInstanceState serve per riportare un eventuale stato dell’attività salvato in precedenza da un’altra istanza che è stata terminata. L’argomento è null nel caso in cui l’attività non abbia uno stato salvato.
- ***protected void onRestart()***
  Richiamato per segnalare che l’attività sta venendo riavviata dopo essere stata precedentemente arrestata.
- ***protected void onStart()***
  Richiamato per segnalare che l’attività sta per diventare visibile sullo schermo.
- ***protected void onResume()***
  Richiamato per segnalare che l’attività sta per iniziare l’interazione con l’utente.
- ***protected void onPause()***
  Richiamato per segnalare che l’attività non sta più interagendo con l’utente.
- ***protected void onStop()***
  Richiamato per segnalare che l’attività non è più visibile sullo schermo.
- ***protected void onDestroy()***
  Richiamato per segnalare che l’applicazione sta per essere terminata.

La prassi richiede che, come prima riga di codice di ciascuno di questi metodi, si richiami l’implementazione di base del metodo che si sta ridefinendo. Ad esempio:

```
protected void onStart() { super.onStart(); // ...}
```

È importante non dimenticare questa regola: le attività sviluppate potrebbero non funzionare!

DESCRIVERE UN’ATTIVITÀ

Dopo che si è creata un’attività, la si deve registrare all’interno del descrittore dell’applicazione (il file *AndroidManifest.xml*), questo affinché il sistema sappia della sua esistenza. Per farlo si usa un tag <*activity*> all’interno del tag <*application*>:

```
<?xml version="1.0" encoding="utf-8"?>
```

ANDROID SDK 1.6

È stata da poco rilasciata la versione 1.6 del kit di sviluppo per Android. Molte sono le novità incluse nella nuova versione: oltre ai consueti bugfix e alle migliorie generali, il sistema comprende ora delle nuove API per il supporto delle gesture e del text-to-speech.
Se siete fermi alla versione 1.5, potete aggiornarvi partendo dall’indirizzo: http://developer.android.com/sdk/
Oltre all’SDK, bisogna aggiornare anche l’ADT (il plug-in per Eclipse).
Farlo è molto semplice: aprite Eclipse e selezionate la voce di menu “*Help » Check for Updates*”, controllate gli aggiornamenti disponibili e selezionate quelli relativi ai componenti Android. Dopo aver scaricato e installato gli aggiornamenti, aprite la maschera della preferenze (voce di menu “*Window » Preferences*”) e alla scheda “*Android*” provvedete ad aggiornare il percorso dell’SDK, in modo da puntare alla versione 1.6.
Con la voce di menu “*Window » Android SDK and AVD Manager*”, create poi un AVD (*Android Virtual Device*) compatibile con la nuova versione del sistema.

```
<manifest xmlns:android="http://schemas.
                              android.com/apk/res/android"
          package=" mypackage.mysubpackage " . . . >
    <application . . . >
        <activity android:name=".MyActivity" . . . >. . .
        </activity>
        . . .
    </application>
</manifest>
```

Con l'attributo *android:name* si specifica il nome della classe registrata come attività. Si può esprimere sia il suo percorso completo (ad esempio *mypackage.mysubpackage.MyActivity*) sia il nome relativo rispetto al package dichiarato nel tag *<manifest>* sovrastante (ad esempio *.MyActivity*).
Altri attributi possono opzionalmente essere inseriti nel tag *<activity>*, allo scopo di meglio dettagliare l'attività che si sta registrando. Tra le tante cose che si possono fare, una delle più importanti è quella di attribuire una *label*, cioè un'etichetta, all'attività. Si tratta, sostanzialmente, di un titolo che il sistema userà per riferirsi all'attività e per presentarla all'utente. L'attributo da utilizzare è *android:label*. Come si è spiegato nel numero precedente, in casi come questo è possibile sia scrivere la stringa direttamente nell'XML sia fare riferimento ad una stringa memorizzata in un file *strings.xml* sotto la directory *res/values*. Nel primo caso, quindi, si userà una formula del tipo:
<activity android:name=".MyActivity" android:label="La mia attività">
Nel secondo caso, invece, si farà alla seguente maniera:
<activity android:name=".MyActivity" android:label="@string/my_activity_label">
Un altro attributo spesso usato con il tag *<activity>* è *android:icon*, che permette di specificare un'icona per l'attività. In questo caso si usa sempre il riferimento ad una immagine presente nella cartella *res/drawable*, qualcosa come:
<activity android:name=".MyActivity" android:icon="@drawable/my_activity_icon">
Se non si specifica alcuna icona, l'attività eredita automaticamente l'icona definita nel sovrastante tag *<application>*. All'interno della coppia di tag *<activity>* … *</activity>*, invece, possono essere allacciate delle relazioni particolari fra l'attività e l'applicazione e fra l'attività ed il sistema. In particolar modo, è possibile collegare all'attività un *intent-filter*:

```
<activity . . .>
  <intent-filter> . . .</intent-filter>
</activity>
```

Nel dizionario di Android, un *intent* è "la descrizione di un'operazione che deve essere eseguita".
Più semplicemente, gli intent sono dei messaggi che il sistema manda a un'applicazione quando si aspetta che questa faccia qualcosa. Di come funzionano gli intent e di cosa si compongono torneremo certamente a parlare in futuro. Per ora ci basta sapere che le attività, attraverso un *intent-filter*, possono essere attivate in risposta ad uno specifico evento. Gli *intent-filter* accettano figli di tre tipi:

- ***<action android:name="nome-azione">***
  Individua gli intent che richiedono l'azione specificata.
- ***<category android:name="nome-categoria">***
  Individua gli intent che appartengono alla categoria specificata.
- ***<data android:mimeType="nome-tipo-mime">***
  <data android:scheme="nome-schema-url">
  Individua gli intent che portano dati del tipo specificato.

Le azioni, le categorie ed i tipi di dato che possono essere usati sono, naturalmente, moltissimi.
Pian piano impareremo a conoscerli. Per ora ci interessa sapere che un'attività dotata di un intent-filter che include l'azione *android.intent.action.MAIN* e la categoria *android.intent.category.LAUNCHER* viene identificata come l'attività principale dell'applicazione. Ciò significa che l'applicazione sarà elencata nel menù di sistema e che, quando l'utente l'avvierà, sarà lanciata proprio l'attività marcata in tale maniera. Ecco perché, in tutti gli esempi dei mesi precedenti, abbiamo sempre usato una formulazione del tipo:

```
<activity android:name=".MyActivity" . . .>
  <intent-filter>
    <action android:name="android.intent.action.MAIN"/>
    <category
   android:name="android.intent.category.LAUNCHER" />
  </intent-filter>
</activity>
```

Oltre all'attività principale, possiamo registrare in *AndroidManifest.xml* quante attività vogliamo, da lanciare secondo necessità, come vedremo tra due paragrafi.

IL PROGETTO ACTIVITYDEMO

Fermiamoci un attimo con la teoria e mettiamo insieme in un esempio pratico i concetti appresi finora. Lo scopo è dimostrare il ciclo di vita delle attività, attraverso una activity che registri in un log tutti i suoi cambi di stato. Creiamo il progetto *ActivityDemo*, con package di riferimento *it.iopro-*

NOTA

PROGRAMMA ANDROID IN C++

Una novità collegata al rilascio della versione 1.6 dell'Android SDK è la possibilità di scrivere applicazioni in C/C++, compilate poi in codice nativo. Nella maggior parte dei casi, comunque, rimane conveniente programmare Android in Java, poiché la piattaforma è stata concepita per essere programmata in questa maniera. Esistono dei casi, tuttavia, in cui si deve agire a basso livello, scavalcando Java e la Dalvik Virtual Machine. Adesso è possibile farlo scaricando ed installando l'Android NDK (*Native Development Kit*), un addendum dell'SDK disponibile al medesimo indirizzo di quest'ultimo.

grammo.activitydemo, ed inseriamo al suo interno la seguente omonima attività:

```
package it.ioprogrammo.activitydemo;
import android.app.Activity; import android.os.Bundle;
import android.util.Log;
public class ActivityDemo extends Activity
{
@Override
 protected void onCreate(Bundle savedInstanceState)
 {
     super.onCreate(savedInstanceState);
     Log.i("ActivityDemo", "Richiamato
          onCreate() con bundle " + savedInstanceState);
 }
 @Override
 protected void onRestart()
 {
     super.onRestart();
     Log.i("ActivityDemo", "Richiamato onRestart()"); }
 @Override
  protected void onStart() {
     super.onStart();
     Log.i("ActivityDemo", "Richiamato onStart()"); }
  @Override
  protected void onResume() {
      super.onResume();
      Log.i("ActivityDemo", "Richiamato onResume()"); }
  @Override
  protected void onPause() {
     super.onPause();
     Log.i("ActivityDemo", "Richiamato onPause()"); }
  @Override
   protected void onStop() {
      super.onStop();
      Log.i("ActivityDemo", "Richiamato onStop()"); }
   @Override
   protected void onDestroy() {
      super.onDestroy();
      Log.i("ActivityDemo", "Richiamato onDestroy()");
   }
}
```

Questo codice, oltre a mostrarci la prassi corretta da applicare quando si ridefiniscono i metodi che intercettano i cambi di stato dell'attività, ci fa fare conoscenza con la classe *android.util.Log*. Come è facile intuire, la classe contiene dei metodi statici per scrivere nel log di sistema. Il metodo *i()*, usato nell'esempio, salva delle righe di log di livello *INFO*. Altri metodi disponibili sono *v()* per il livello *VERBOSE*, *d()* per il livello *DEBUG*, *w()* per il livello *WARN*, ed *e()* per il livello *ERROR*. Registriamo l'attività nel manifesto dell'applicazione, marcandola come attività principale di lancio:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.
android.com/apk/res/android" package="it.ioprogrammo.
                  activitydemo" android:versionCode="1"
                           android:versionName="1.0">
<application android:icon="@drawable/icon"
                    android:label="@string/app_name">
<activity android:name=
   ".ActivityDemo" android:label="@string/app_name">
<intent-filter>
 <action android:name="android.intent.action.MAIN" />
<category
  android:name="android.intent.category.LAUNCHER" />
</intent-filter>
</activity>
</application>
<uses-sdk android:minSdkVersion="3" />
</manifest>
```

Non dimentichiamo di includere un file *res/values/strings.xml* con le risorse riferite nel manifesto:

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
 <string name="app_name">ActivityDemo LABEL</string>
</resources>
```

L'applicazione è ora pronta per essere avviata. Avviate l'emulatore e fate qualche prova. Mandate l'attività in secondo piano, magari azionando qualche altra applicazione fra quelle disponibili nel sistema. Quindi date un'occhiata ai log emessi, per verificare il ciclo di vita dell'attività. Usando Eclipse i log possono essere consultati nella prospettiva di lavoro chiamata "*DDMS*", nella scheda "*LogCat*".

SOTTO-ATTIVITÀ

Come spiegato in apertura, un'applicazione Android può contenere più di un'attività. In questo caso una soltanto sarà marcata come attività principale di lancio. Le altre saranno, invece, delle sotto-attività, che l'attività principale potrà lanciare quando ce n'è bisogno. Realizzare una sotto-attività è semplice tanto quanto realizzare l'attività principale: ancora una volta è sufficiente estendere *android.app.Activity*.
Le attività secondarie vanno poi registrate nel file *AndroidManifest.xml*, senza però applicare l'intent-filter con l'azione e la categoria usate invece dall'attività principale. L'attività principale può lanciare delle sotto-attività ricorrendo al metodo *startActivity()*. Questo accetta come argomento un oggetto di tipo android.content.Intenet che, come è facile intuire, rappresenta un intent. Con questo intento bisogna esprimere quale sotto-attività deve essere avviata. Immaginiamo di voler lanciare la sotto-attività rappresentata dalla

ALTRI ATTRIBUTI DEL TAG <ACTIVITY>

Il tag <*activity*>, nel manifest dell'applicazione, ammette una grande varietà di attributi, oltre a quelli citati nell'articolo. Ad esempio con *android: excludeFromRecents* si può fare in modo che l'attività non venga presentata nell'elenco delle attività lanciate di recente, con *android:screen Orientation* si può forzare un certo orientamento dello schermo e con *android:theme* si può cambiare il tema grafico usato dall'attività. Un elenco completo è compreso nella documentazione ufficiale, all'indirizzo: http://developer.android.com/guide/topics/manifest/activity-element.html

classe *MySubActivity*. Tutto quello che dovremo fare, dall'interno dell'attività principale, è formulare un'istruzione del tipo:

startActivity(new Intent(this, MySubActivity.class));

La sotto-attività verrà lanciata e prenderà lo schermo al posto di quella principale.

LogCat

Log

Time		pid	tag	Message
10-08 12:30:01.183	I	721	ActivityDemo	Richiamato onDestroy()
10-08 12:30:13.894	I	721	ActivityDemo	Richiamato onCreate() con bundle null
10-08 12:30:13.894	I	721	ActivityDemo	Richiamato onStart()
10-08 12:30:13.894	I	721	ActivityDemo	Richiamato onResume()
10-08 12:30:22.613	I	721	ActivityDemo	Richiamato onPause()
10-08 12:30:22.813	I	721	ActivityDemo	Richiamato onStop()
10-08 12:30:22.813	I	721	ActivityDemo	Richiamato onDestroy()
10-08 12:30:31.324	I	721	ActivityDemo	Richiamato onCreate() con bundle null
10-08 12:30:31.324	I	721	ActivityDemo	Richiamato onStart()
10-08 12:30:31.324	I	721	ActivityDemo	Richiamato onResume()
10-08 12:30:39.623	I	721	ActivityDemo	Richiamato onPause()
10-08 12:30:39.953	I	721	ActivityDemo	Richiamato onStop()

Filter: tag:ActivityDemo

***Fig. 2:** Il log di ActivityDemo permette di osservare il ciclo di vita dell'attività*

SUBACTIVITYDEMO

Mettiamo in pratica quanto descritto nel paragrafo precedente. Il progetto si chiama *SubActivityDemo*, e il package di riferimento è *it.ioprogrammo.subactivitydemo*. All'interno di questo inseriremo due attività: *MainActivity* e *SubActivity*. La prima sarà poi registrata come attività principale e conterrà un tasto per avviare la seconda. *SubActivity* conterrà, invece, un tasto per chiuderla e passare all'attività principale:

```
package it.ioprogrammo.subactivitydemo;
import android.app.Activity;
import android.content.Intent;
import android.os.Bundle; import android.view.View;
import android.widget.Button;
public class MainActivity extends Activity {
@Override
public void onCreate(Bundle savedInstanceState) {
   super.onCreate(savedInstanceState);
   Button button = new Button(this);
   button.setText("Lancia SubActivity");
   button.setOnClickListener(new View.OnClickListener() {
   @Override
   public void onClick(View v) { startSubActivity(); }});
      setContentView(button); }
private void startSubActivity() {
  Intent intent = new Intent(this, SubActivity.class);
  startActivity(intent);
}}
```

Quanto spiegato nel paragrafo precedente è stato messo in pratica all'interno del metodo *startSub Activity()*. Studiamo il codice di *SubActivity*:

```
package it.ioprogrammo.subactivitydemo;
import android.app.Activity; import android.os.Bundle;
import android.view.View;import android.widget.Button;
public class SubActivity extends Activity
{
 @Override
 protected void onCreate(Bundle savedInstanceState){
   super.onCreate(savedInstanceState);
   Button button = new Button(this);
   button.setText("Termina SubActivity");
   button.setOnClickListener(new View.OnClickListener()
   {
     @Override
     public void onClick(View v) {finish();}});
     setContentView(button); }
}
```

Dal codice apprendiamo una nozione nuova: un'attività, sia principale sia secondaria, può terminare e chiudersi spontaneamente invocando il proprio metodo *finish()*. Registriamo adesso le due attività nel descrittore dell'applicazione:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.
                    android.com/apk/res/android"
 package="it.ioprogrammo.subactivitydemo"
                          android:versionCode="1"
 android:versionName="1.0">
 <application android:icon="@drawable/icon"
                   android:label="@string/app_name">
 <activity android:name=".MainActivity"
    android:label="@string/main_activity_label">
    <intent-filter>
    <action android:name="android.intent.action.MAIN" />
    <category
  android:name="android.intent.category.LAUNCHER" />
</intent-filter>
</activity>
<activity android:name=".SubActivity"
 android:label="@string/sub_activity_label"></activity>
</application>
<uses-sdk android:minSdkVersion="3" />
</manifest>
```

Per completare ecco il file *res/values/strings.xml*:

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
<string name="app_name">ManyActivitiesDemo</string>
<string name="main_activity_label">Main Activity</string>
<string name="sub_activity_label">Sub Activity</string>
</resources>
```

Non resta che eseguire l'esempio e provare il lancio e la chiusura della sotto-attività.

Carlo Pelliccia

L'AUTORE

Carlo Pelliccia lavora presso 4IT (www.4it.it), dove si occupa di analisi e sviluppo software per piattaforme Java. Nella sua carriera di technical writer ha pubblicato cinque manuali ed oltre centocinquanta articoli, molti dei quali proprio tra le pagine di ioProgrammo. Il suo sito, che ospita anche diversi progetti Java Open Source, è disponibile all'indirizzo www.sauronsoftware.it

INTERFACCE: LAYOUT E COMPONENTI

QUARTO APPUNTAMENTO. INIZIA LA TRATTAZIONE DEI CONCETTI E DEGLI STRUMENTI DI ANDROID PER LA COSTRUZIONE E LA GESTIONE DELLE INTERFACCE UTENTE. SI COMINCIA CON I WIDGET ED I LAYOUT DI BASE, INDISPENSABILI IN OGNI APPLICAZIONE

Dopo aver scoperto cosa sono e come funzionano le attività, studiamo ora come si costruiscono e si gestiscono le interfacce utente in Android. Apprenderemo le pratiche utili per costruire oggetti grafici in grado di interagire con chi impugna il device. Si tratta di un passaggio cruciale del nostro percorso di apprendimento: tutti i dispositivi mobili di nuova generazione puntano tantissimo sull'interazione con l'utente.

ANDROID 2.0 ED IL NUOVO SDK

Collegatevi all'indirizzo di riferimento per il kit di sviluppo di Android, cioè:

http://developer.android.com/sdk/

Da qui scaricare il più recente SDK disponibile per la vostra piattaforma (al momento in cui questo articolo viene scritto, la versione più recente è la *r3*). Estraete l'archivio scaricato sul vostro disco rigido, al percorso che preferite. Con la riga di comando, adesso, posizionatevi sulla directory *tools* del kit di sviluppo. Lanciate il comando:

android

Verrà caricata una GUI che vi permette di gestire i device virtuali ed i componenti aggiuntivi. Il nuovo SDK, appena scaricato e installato, è praticamente vuoto. Contiene solo gli strumenti di sviluppo, e al suo interno non c'è traccia di alcuna versione del sistema. La prima cosa da fare, quindi, è selezionare la voce "*Available Packages*", che permette di visionare ed installare i componenti aggiuntivi, tra i quali ci sono anche le differenti versioni della piattaforma. Una volta selezionata la voce, espandete l'elenco dei componenti disponibili per l'unica fonte inizialmente registrata. Selezionate le piattaforme con le quali intendete sviluppare (sicuramente la nuova 2.0) ed i componenti aggiuntivi che giudicate utili (come la documentazione). Se volete, potete anche selezionare tutto, in modo da disporre di più versioni del sistema per sviluppare e testare le vostre applicazioni. Azionando il tasto "Install Selected" vi verrà chiesto di accettare le licenze dei componenti selezionati. Usate l'opzione "*Accept All*" e proseguite poi con il tasto "*Install Accepted*". Il tool procederà al download e all'installazione dei componenti richiesti. Naturalmente potrete ripetere questa operazione in futuro, per controllare il rilascio di nuove versioni del sistema e per configurarle nel vostro SDK. Dopo aver concluso il download e l'installazione, non dimenticatevi di creare uno o più device virtuali (AVD) per i vostri sviluppi (voce "*Virtual Devices*" nel tool) compatibili con Android 2.0 e con le altre versioni che avete scaricato. Se sviluppate con Eclipse, come suggerito in questo corso, dovrete anche aggiornare l'ADT e collegarlo alla nuova installazione dell'SDK.

VIEW E VIEWGROUP

Andiamo all'argomento del giorno, cioè alle interfacce grafiche e ai componenti che le attività possono usare per interagire con l'utente. I primi due concetti che dobbiamo assorbire si chiamano *View* e *ViewGroup*, e corrispondono alla maniera di Android di classificare ed organizzare ciò che è sullo schermo. I bottoni, i campi di testo, le icone e tutti gli altri congegni di un'interfaccia grafica sono oggetti *View*. I *ViewGroup*, invece, sono dei contenitori che possono mettere insieme più oggetti *View*. I *ViewGroup*, inoltre, sono a loro volta degli oggetti *View*, e di conseguenza un possono contenere altri *ViewGroup*. Grazie a questa intuizione è possibile organizzare i componenti sullo schermo secondo uno schema ad albero, come quello di Fig.2.

I componenti *View* estendono tutti la classe base *android.view.View*. Nella libreria standard di Android ci sono già molti componenti di questo tipo, soprattutto nel pacchetto *android.widget*.

Oltre ai widget di base, ad ogni modo, è sempre possibile estendere la classe *View* e realizzare i propri componenti custom. Il più delle volte non c'è bisogno di farlo, poiché quelli forniti da Android bastano per tutte le principali necessità. È comunque importante che sia data questa possibilità. La classe *android.view.ViewGroup* è una speciale estensione di *View*. Come accennato in precedenza, e come rappresentato in figura, un *ViewGroup* è una speciale *View* che può contenere altre *View*. Per questo motivo gli oggetti *ViewGroup* dispongono di diverse implementazioni del metodo *addView()*, che permette proprio di aggiungere una nuova *View* al gruppo:

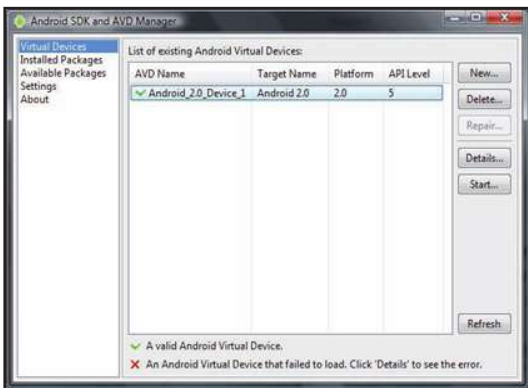



Fig. 1: Dopo il download della nuova versione della piattaforma si deve configurare un device virtuale che la installi, altrimenti non sarà possibile usarla per lo sviluppo

- **public void addView(View child)**
  Aggiunge un oggetto *View* al gruppo.

- **public void addView(View child, int index)**
  Aggiunge un oggetto *View* al gruppo, specificandone la posizione attraverso un indice (*index*).

- **public void addView(View child, int width, int height)** Aggiunge un oggetto *View* al gruppo, specificandone larghezza (*width*) ed altezza (*height*).

- **public void addView(View child, View Group.LayoutParams params)**
  Aggiunge un oggetto *View* al gruppo, applicando una serie di parametri di visualizzazione ed organizzazione del componente (*params*).

- **public void addView(View child, int index, ViewGroup.LayoutParams params)**
  Aggiunge un oggetto *View* al gruppo, specificando la posizione attraverso un indice (*index*) ed applicando una serie di parametri di visualizzazione ed organizzazione del componente (*params*).

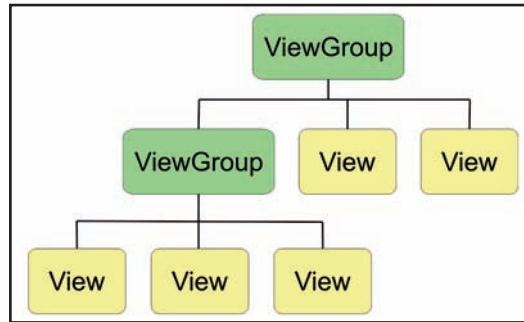



Fig. 2: Android organizza i componenti sullo schermo attraverso i componenti di base View e ViewGroup

ViewGroup è una classe astratta. Pertanto non può essere istanziata direttamente. Come nel caso di *View*, è possibile realizzare il proprio *ViewGroup* custom, ma il più delle volte conviene scegliere fra le tante implementazioni messe a disposizione dalla libreria Android. Queste implementazioni differiscono nella maniera di presentare i componenti che sono al loro interno: alcuni li mettono uno dopo l'altro, altri li organizzano in una griglia, altri ancora possono essere usati per avere una gestione a schede dello schermo, e così via. Ovviamente conosceremo presto tutte le principali implementazioni di *ViewGroup*.

Una volta che, combinando oggetti *View* e *ViewGroup*, si è ottenuta l'interfaccia utente che si desidera, è necessario che questa venga mostrata sullo schermo. Come abbiamo scoperto mediante alcuni esempi preliminari, le attività (cioè gli oggetti *android.app.Activity*) mettono a disposizione un metodo *setContentView()*, disponibile nelle seguenti forme:

- **public void setContentView(View view)**
  Mostra sullo schermo l'oggetto *View* specificato.
- **public void setContentView(View view, ViewGroup.LayoutParams params)**
  Mostra sullo schermo l'oggetto *View* specificato, applicando una serie di parametri di visualizzazione ed organizzazione del componente (*params*).

WIDGET

Con il termine *widget* (*congegno*) si indicano quei componenti di base per l'interazione con l'utente, come i bottoni, le check box, le liste, i campi di testo e così via. I widget predefiniti di Android estendono tutti (direttamente o indirettamente) la classe *View*, e sono conservati nel package *android.widget*. Esaminiamone alcuni in una veloce panoramica:

NOTA

PROBLEMI SSL

Se, nel tentativo di scaricare gli aggiornamenti automatici dell'SDK, vi doveste imbattere in un errore SSL, potete risolverlo alla seguente maniera: nell'elenco di sinistra scegliete la voce "*Settings*", attivate la *checkbox "Force https://... sources to be fetched using http://...*", confermate con il tasto "*Save & Apply*". Adesso tornate sulla scheda "*Available Packages*" e tentate di nuovo il download degli aggiornamenti.

Fig. 3: TextView

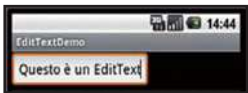
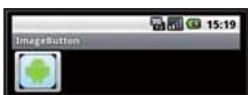
ANDROID.WIDGET.TEXTVIEW
Permette di mostrare del testo all'utente. Il messaggio da visualizzare può essere impostato con il metodo *setText()*, che può accettare come parametro sia una stringa sia un riferimento a risorsa preso dal gruppo *R.string* (cfr. ioProgrammo 144).

ANDROID.WIDGET.EDITTEXT
Estende *TextView* e permette all'utente di modificare il testo mostrato. Il testo digitato può essere recuperato con il metodo *getText()*, che restituisce un oggetto del tipo *android.text.Editable*. Gli oggetti *Editable* sono simili alle stringhe, ed infatti implementano l'interfaccia *java.lang.Char Sequence*.

Fig. 4: EditText

ANDROID.WIDGET.BUTTON
Realizza un bottone che l'utente può premere o cliccare. Il componente espande *TextView*, e per questo è possibile impostare il testo mostrato al suo interno con il metodo *setText()*, sia con parametro stringa sia con riferimento a risorsa del gruppo *R.string*.

Fig. 5: Button

ANDROID.WIDGET.IMAGEVIEW
Un componente che permette di mostrare un'immagine. Metodi utili sono: *setImage Bitmap()*, che accetta un oggetto di tipo *android.graphics.Bitmap*; *setImageDrawable()*, che accetta un argomento *android. graphics. drawable. Drawable*; *setImage Resource()*, che accetta un riferimento a risorsa drawable.

Fig. 6: ImageView

ANDROID.WIDGET.IMAGEBUTTON
Un bottone con un'immagine. Estende *Image View*, e quindi espone gli stessi metodi di quest'ultima per impostare l'immagine mostrata.

Fig. 7: ImageButton

ANDROID.WIDGET.CHECKBOX
Questo componente realizza una casella di spunta (*check box*, appunto). Estende *Button* e *TextView*, pertanto il testo a fianco della casella può essere impostato con i metodi *setText()* già noti..

Fig. 8: CheckBox

ANDROID.WIDGET.RADIOBUTTON
Questo componente realizza un bottone radio. Come nel caso di CheckBox, le classi base *Button* e *TextView* forniscono i metodi necessari per l'impostazione del testo visualizzato. Un bottone radio, da solo, non ha senso. Due o più bottoni radio, pertanto, possono essere raggruppati all'interno di un *android.widget.RadioGroup*. L'utente, così, potrà attivare soltanto una delle opzioni del gruppo.

Fig. 9: RadioButton

ANDROID.WIDGET.TOGGLEBUTTON
Un bottone “ad interruttore”, che può essere cioè “on” o “off”. Può essere usato per far attivare o disattivare delle opzioni.

ANDROID.WIDGET.DATEPICKER
Un componente che permette di scegliere una data selezionando giorno, mese ed anno. La data impostata dall'utente può essere recuperata servendosi dei metodi *getDayOfMonth()*, *getMonth()* e *getYear()*.

ANDROID.WIDGET.TIMEPICKER
Un componente che permette di scegliere un orario selezionando ora e minuto. L'orario impostato dall'utente può essere recuperato servendosi dei metodi *getCurrentHour()* e *getCurrent Minute()*.

ANDROID.WIDGET.ANALOGCLOCK
Un componente che mostra all'utente un orologio analogico.

ANDROID.WIDGET.DIGITALCLOCK
Un componente che mostra all'utente un orologio digitale.

Tutti gli oggetti discussi finora richiedono, nei loro costruttori, un oggetto che estenda la classe astratta *android.content.Context*. Si tratta di una struttura che permette l'accesso al sistema e che costituisce il contesto di esecuzione dell'applicazione. Non dovete preoccuparvi di come ottenere oggetti di questo tipo: *android.app.Activity* estende indirettamente *Context*, per cui dall'interno di un'attività, vi sarà sufficiente usare la parola chiave this. Ad esempio:

```
Button b = new Button(this);
```

La considerazione vale per le attività, ma anche per tanti altri contesti della programmazione Android: più o meno tutte le classi che sono mattoni fondamentali del sistema estendono direttamente o indirettamente la classe astratta *android.content.Context*.
Sul CD-Rom allegato alla rivista trovate, insieme con i codici di questo articolo, gli esempi d'uso di ciascuno dei widget citati.

LAYOUT

Con il termine *layout* (*disposizione*, *impaginazione*), in Android, si identificano tutti quei *ViewGroup* utilizzabili per posizionare i widget sullo schermo. Android fornisce una serie di layout predefiniti. Esaminiamone alcuni.

android.widget.FrameLayout
Il più semplice e basilare dei layout: accetta un widget, lo allinea in alto a sinistra e lo estende per tutta la dimensione disponibile al layout stesso. Ecco un semplice esempio di utilizzo, che allarga un bottone all'intera area a disposizione

di un'attività:

```
Button button = new Button(this);
button.setText("Bottone");
FrameLayout layout = new FrameLayout(this);
layout.addView(button);
setContentView(layout);
```

android.widget.RelativeLayout
Come *FrameLayout*, vuole un solo componente al suo interno, ma a differenza di quest'ultimo, lo disegna nelle sue dimensioni ideali, senza allargarlo per ricoprire l'intera area a disposizione. Per default, il componente viene allineato in alto a sinistra, ma è possibile controllare l'allineamento servendosi del metodo *setGravity()*. Questo accetta un argomento di tipo *int*, che è bene scegliere fra le costanti messe a disposizione nella classe *android.view.Gravity*. I valori possibili, nel caso di *RelativeLayout*, sono i seguenti:

- *Gravity.**TOP*** allinea il widget in alto.
- *Gravity.**BOTTOM*** allinea il widget in basso.
- *Gravity.**LEFT*** allinea il widget a sinistra.
- *Gravity.**RIGHT*** allinea il widget a destra.
- *Gravity.**CENTER_HORIZONTAL*** allinea il widget al centro orizzontalmente.
- *Gravity.**CENTER_VERTICAL*** allinea il widget al centro verticalmente.
- *Gravity.**CENTER*** allinea il widget al centro sia orizzontalmente sia verticalmente.

Più costanti *Gravity*, purché non in contrasto fra di loro, possono essere concatenate in un solo valore servendosi dell'operatore binario OR (che in Java si rende con il simbolo |, detto *pipe*). Ad esempio per allineare in basso a destra si scrive:

Gravity.BOTTOM | Gravity.RIGHT

Ecco un campione di codice che dimostra l'uso di un *RelativeLayout* all'interno di un'attività:

```
Button button = new Button(this);
button.setText("Bottone");
RelativeLayout layout = new RelativeLayout(this);
layout.setGravity(Gravity.TOP |
                                Gravity.CENTER_HORIZONTAL);
layout.addView(button);
setContentView(layout);
```

ANDROID.WIDGET.LINEARLAYOUT
Un layout utile per disporre più componenti uno di seguito all'altro, sia orizzontalmente sia verticalmente. Una volta creato il layout, il suo orientamento può essere stabilito chiamando il metodo setOrientation(), con argomento pari a *LinearLayout.HORIZONTAL* o *LinearLayout.VERTICAL*. Con l'orientamento orizzontale i componenti verranno messi tutta sulla stessa riga, uno dopo l'altro. Con l'allineamento verticale, invece, si procede lungo una colonna, e quindi i widget saranno uno sopra l'altro.
Esaminiamo il caso dell'allineamento orizzontale. In questo caso i componenti vengono introdotti lungo una sola linea. Il sistema accetta di aggiungere componenti finché c'è spazio. Se si va di poco oltre la dimensione della riga, il sistema tenta un aggiustamento restringendo i componenti al di sotto delle loro dimensioni ideali. Raggiunto un certo limite, comunque, il sistema si rifiuta di andare oltre, ed i componenti di troppo non saranno più visualizzati. Il metodo *setGravity()*, nell'allineamento orizzontale, può essere usato per decidere dove posizionare e come organizzare la riga dei componenti rispetto allo spazio disponibile.
Ecco un esempio:

```
Button button1 = new Button(this);
button1.setText("Bottone 1");
Button button2 = new Button(this);
button2.setText("Bottone 2");
Button button3 = new Button(this);
button3.setText("Bottone 3");
LinearLayout layout = new LinearLayout(this);
layout.setOrientation(LinearLayout.HORIZONTAL);
layout.setGravity(Gravity.CENTER_HORIZONTAL);
layout.addView(button1);
layout.addView(button2);
layout.addView(button3);
setContentView(layout);
```

Nei *LinearLayout* verticali i componenti vengono aggiunti uno sopra all'altro, ed espansi in orizzontale fino ad occupare tutto lo spazio a disposizione del layout. In questo caso *setGravity()* può essere usato per decidere se allineare la colonna in alto, in basso o al centro. Il sistema aggiunge componenti finché c'è spazio nella colonna. Superato il limite, i componenti di troppo non vengono visualizzati. Ecco un esempio:

```
Button button1 = new Button(this);
button1.setText("Bottone 1");
Button button2 = new Button(this);
button2.setText("Bottone 2");
Button button3 = new Button(this);
button3.setText("Bottone 3");
...
```

La classe *android.widget.RadioGroup*, presentata sopra e utile per mettere insieme più *RadioButton*, estende *LinearLayout* e gode pertanto di tutte le proprietà appena mostrate.

Fig. 10: ToggleButton

Fig. 11: DatePicker

Fig. 12: TimePicker

Fig. 13: AnalogClock

Fig. 14: DigitalClock

Fig. 15: Un bottone, inserito in un FrameLayout, viene espanso fino alla dimensione massima del layout

Fig. 16: RelativeLayout utilizzato per disporre un bottone, nelle sue dimensioni ideali

Fig. 17: Un LinearLayout che allinea tre bottoni in orizzontale, al centro

Fig. 18: Un TableLayout con nove bottoni disposti su tre righe e tre colonne

Fig. 19: Una maschera di immissione dati realizzata combinando più widget e più layout

ANDROID.WIDGET.TABLELAYOUT

Un layout che permette di sistemare i componenti secondo uno schema a tabella, suddiviso cioè in righe e colonne. I *TableLayout* vanno costruiti aggiungendo al loro interno degli oggetti *TableRow*, ciascuno dei quali forma una riga della tabella.

Ogni riga è suddivisa in colonne. In ciascuna cella può essere inserito un componente. La gravità, cioè il metodo *setGravity()*, può essere usato sia su *TableLayout* che su *TableRow*, per stabilire gli allineamenti relativi. Ecco un semplice esempio di tre righe e tre colonne:

```
int rows = 3;
int columns = 3;
TableLayout layout = new TableLayout(this);
layout.setGravity(Gravity.CENTER);
for (int i = 0; i < rows; i++) {
    TableRow tableRow = new TableRow(this);
    tableRow.setGravity(Gravity.CENTER);
    for (int j = 0; j < columns; j++)
    {
            Button button = new Button(this);
            button.setText("Bottone " +
                                ((columns * i) + j + 1));
            tableRow.addView(button);
    }
  layout.addView(tableRow);
}
setContentView(layout);
```

Si faccia attenzione al fatto che, se la tabella eccede le dimensioni a disposizione, una parte di essa non sarà visibile. Su come Android ripartisca la dimensione da assegnare a ciascuna colonna, si può agire con i seguenti metodi:

- *public void* ***setColumnCollapsed****(int column Index, boolean isCollapsed)*
  Stabilisce se una colonna è *collapsed*. Quando una colonna è *collapsed*, non viene mostrata sullo schermo.

- *public void* ***setColumnShrinkable****(int column Index, boolean isShrinkable)*
  Stabilisce se una colonna è *shrinkable*. Quando una colonna è *shrinkable*, il sistema cerca di restringerla il più possibile, per fare in modo che occupi poco spazio.

- *public void* ***setColumnStretchable****(int column Index, boolean isStretchable)*
  Stabilisce se una colonna è *stretchable*. Quando una colonna è *stretchable*, il sistema tende ad allargarla fornendogli lo spazio extra di cui dispone.

Si faccia attenzione al fatto che gli indici delle colonne, in Android, partono da 0. *TableLayout* dispone inoltre di alcuni metodi di comodo che permettono, in un sol colpo, di applicare le medesime impostazioni di *shrink* o di *stretch* a tutte le colonne. Questi metodi sono *setShrinkAll Columns()* e *setStretchAllColumns()*.

Sul CD-Rom allegato alla rivista trovate, insieme con i codici di questo articolo, gli esempi d'uso di ciascuno dei layout citati.

METTERE INSIEME WIDGET E LAYOUT

I widget ed i layout illustrati sinora, naturalmente, devono essere combinati in maniera coerente. I layout, in maniera particolare, possono e devono essere annidati l'uno dentro l'altro, finché non si ottiene il design desiderato. Proviamo insieme con un esempio semplice ed efficace. Facciamo il caso che dobbiamo realizzare, in un'attività, una maschera di input attraverso la quale l'utente può specificare il proprio nome, il proprio cognome ed il proprio sesso. Al termine dell'operazione, l'utente può salvare i dati con un tasto "*Salva*" o annullarli con il bottone "*Annulla*". Ecco il codice necessario per realizzare quanto teorizzato:

```
TextView label1 = new TextView(this);
label1.setText("Nome:");
EditText edit1 = new EditText(this);
TextView label2 = new TextView(this);
label2.setText("Cognome:");
EditText edit2 = new EditText(this);
...
```

Si sono adoperati dei widget *TextView*, per le etichette, *EditText*, per i campi ad immissione libera, RadioButton (con *RadioGroup*), per la selezione tra un elenco di opzioni, e *Button*, per i bottoni di azione finali. I componenti sono stati disposti sullo schermo annidando diversi tipi di layout. I campi del modulo sono stati messi insieme servendosi di un *TableLayout*, che dispone le etichette sulla colonna di sinistra ed i widget manipolabili su quella di destra. La pulsantiera con i bottoni "*Salva*" e "*Annulla*" è stata invece realizzata servendosi di un *LinearLayout* orizzontale, che affianca e centra i due widget. I due layout, infine, sono stati messi insieme servendosi di un terzo contenitore, nello specifico un *Linear Layout* verticale, che ha disposto la tabella in alto e la pulsantiera sotto di questa. Tutto, infine, è stato centrato sullo schermo. Il risultato ottenuto è mostrato in figura.

Carlo Pelliccia

INTERFACCE IN XML PER ANDROID

QUINTO APPUNTAMENTO. VI È SEMBRATO CHE IL DESIGN JAVA DI UN'INTERFACCIA UTENTE, IN ANDROID, SIA LUNGO E NOIOSO? NESSUN PROBLEMA! OGGI IMPAREREMO A SERVIRCI DELL'XML PER VELOCIZZARE E SEMPLIFICARE L'OPERAZIONE

Lo scorso mese abbiamo imparato a disporre sullo schermo i principali widget messi a disposizione da Android: bottoni, caselle di testo, check box e via discorrendo. La logica che permea la loro creazione ed il loro utilizzo, come abbiamo potuto osservare, non si discosta di molto da quella adottata da altre librerie Java per le interfacce utente, come AWT e Swing. Questo modo di creare le GUI è potente, ma anche estremamente tedioso. Ogni volta che si deve utilizzare un widget, lo si deve creare, personalizzare ed inserire in un contenitore predisposto in precedenza. Sin dalle origini delle interfacce basate sui widget, i creatori delle piattaforme di sviluppo hanno cercato di porre rimedio a questo difetto. Nella maggior parte dei casi si è fatto ricorso ad editor visuali: il programmatore, anziché scrivere codice, trascina i componenti sull'editor, dimensionandoli ad occhio ed impostandone le caratteristiche salienti mediante delle procedure guidate. Il lavoro sporco lo fa l'editor in sottofondo, generando ed interpretando il codice di programmazione necessario. Questo approccio è valido, ma da solo non costituisce una vera e propria soluzione al problema. Il codice prolisso e difficile da gestire, infatti, è ancora lì: l'ambiente di sviluppo non ha fatto altro che nascondere la sporcizia sotto il tappeto. Gli editor visuali, poi, sono molto difficili da realizzare, perché devono interpretare e generare del codice complesso, ed infatti ne esistono davvero pochi di buona qualità. Il codice generato automaticamente, infine, è spesso un obbrobrio. L'ambiente, infatti, non ha l'intelligenza sufficiente per scrivere e mantenere un codice leggibile e performante.
Con l'avvento dei browser moderni, di AJAX e degli interpreti di nuova concezione, si sono portate sul Web molte applicazioni che, fino a ieri, erano appannaggio esclusivo degli ambienti desktop e dei linguaggi compilati. I client di posta elettronica, ad esempio, stanno scomparendo a favore delle webmail. Il proliferare delle applicazioni Web sta facendo maturare velocemente gli strumenti di sviluppo propri di questo ambito. All'inizio la programmazione Web sottraeva idee alla programmazione delle applicazioni native, emulandone approcci e strumenti; ora, invece, si assiste ad un'inversione di tendenza. La programmazione Web, ad esempio, ha dimostrato quanto sia più facile gestire un'interfaccia utente descrivendone i componenti con un linguaggio a marcatori, anziché con un linguaggio di programmazione.
I linguaggi a marcatori come HTML ed XML ben si prestano a questo genere di operazioni: sono più facili da leggere e da scrivere, sia per l'uomo sia per la macchina (cioè per gli editor visuali). Così oggi le piattaforme moderne applicano alla programmazione di applicazioni native il medesimo principio, fornendo agli sviluppatori framework ed editor basati perlopiù su XML. Uno dei primi tentativi in tal senso ad aver avuto successo è stato *XUL*, nato in casa Mozilla ed impiegato per le GUI di Firefox e di altre applicazioni della fondazione, poi importato anche in altri ambiti ed ambienti. Mi ricollego ora al discorso precedente, ripetendo per l'ennesima volta che Android è un sistema operativo di moderna concezione. Come abbiamo imparato il mese scorso, le GUI possono essere realizzate con un approccio classico, basato sul codice di programmazione. Spesso e volentieri, però, è più semplice ricorrere all'approccio moderno, basato su XML, che Android rende disponibile nativamente.

LAYOUT XML

Veniamo al dunque. Avrete sicuramente notato che la struttura predefinita di un progetto Android creato in Eclipse contiene sempre la directory *res/layout*. Abbiamo già conosciuto alcune fra le sottodirectory di *res* e, in tutti i casi, abbiamo osservato come la piattaforma di sviluppo gestisse in maniera speciale le differenti categorie di risorse possibili. La cartella *layout* non fa eccezione. Al suo interno possono essere disposti dei file XML che il sistema interpreterà come descrizioni dichiarative dei layout e dei widget che saranno poi usati in una o più attività dell'applicazione. Un esempio, in questo caso, vale più di mille parole. Il mese scorso abbiamo chiuso con questo codice:

```
TextView label1 = new TextView(this);
label1.setText("Nome:");
EditText edit1 = new EditText(this);
TextView label2 = new TextView(this);
label2.setText("Cognome:");
...
```

Si tratta del codice necessario per assemblare un modulo per l'inserimento dei dati anagrafici di una persona (nome, cognome, sesso). Era un buon espediente per mostrare l'utilizzo combinato di alcuni layout (*LinearLayout, TableLayout*) ed alcuni widget (*TextView, EditText, Button, RadioButton*).
L'esempio ha l'effetto collaterale di dimostrare come sia poco naturale usare Java per assemblare un'interfaccia utente: il codice è poco leggibile, i nomi delle variabili poco significativi, e bisogna concentrarsi molto per capire chi contiene cosa. È il classico caso in cui il formato XML è più vantaggioso:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
xmlns:android="http://schemas.android.com/apk/res
                                          /android"
        android:layout_width="fill_parent"
        android:layout_height="fill_parent"
        ...
```

Questo XML, dopo averci fatto l'occhio, risulta molto più semplice da gestire del suo corrispettivo Java. Anzitutto, per quanto riguarda la composizione dei layout, l'utilizzo dell'indentatura permette di individuare immediatamente chi è il contenitore e chi il contenuto. Gli attributi di XML, poi, sono molto più semplici ed intuitivi, rispetto ai metodi del tipo *setProprietà()* di Java. Con gli attributi è più semplice impostare le proprietà di ogni singolo componente, come il testo visualizzato, il padding, la gravità e così via. Creare un editor visuale in grado di leggere e scrivere questo XML, inoltre, è estremamente più facile che realizzare un editor in grado di fare lo stesso con del codice Java. In Eclipse, insieme con il plug-in ADT per lo sviluppo dei progetti Android, ne avete già installato uno. Provate a creare un file di layout in un progetto Android, sotto il percorso *res/layout*. Facendo doppio clic sul file, l'ambiente lo aprirà nel suo editor visuale. Qui è possibile aggiungere layout e widget semplicemente pescandoli dal menu sulla sinistra e trascinandoli sulla schermata al centro, che rappresenta l'interfaccia grafica così come apparirà nello smartphone. Selezionando un componente è possibile accedere all'elenco delle sue proprietà, mostrate nella scheda "*Properties*" in basso. Da qui è possibile manipolare i parametri del componente. Con un po' di pratica si riesce a costruire velocemente qualsiasi tipo di interfaccia. È comunque possibile lavorare "a mano" sul codice XML, attivando la linguetta in basso che riporta il nome del file aperto. L'ambiente, anche in questo caso, fornisce alcune utili agevolazioni, come l'auto-completamento dei tag, degli attributi e dei valori, con tanto di aiuto in linea, come mostrato in figura.

RICHIAMARE UN LAYOUT XML

Nella directory *res/layout* si possono memorizzare quanti file si desidera. L'ambiente li compila e genera automaticamente un riferimento verso ciascuno di essi nella classe *R*, all'interno gruppo *layout*. Ad esempio il file *res/layout/mioLayout.xml* avrà il suo riferimento in *R.layout.mioLayout*. Questo riferimento, passando al codice Java, può essere utilizzato per invocare e adoperare il layout realizzato. La classe *Activity*, ad esempio, dispone di una versione di *setContentView()* che accetta come argomento un riferimento ad un oggetto di tipo layout. Continuando con l'esempio XML del paragrafo precedente (che va salvato al percorso *res/layout/main.xml*), realizziamo un'attività in grado di caricare e mostrare il layout realizzato:

```
package it.ioprogrammo.xmllayoutdemo1;
import android.app.Activity;
import android.os.Bundle;
```

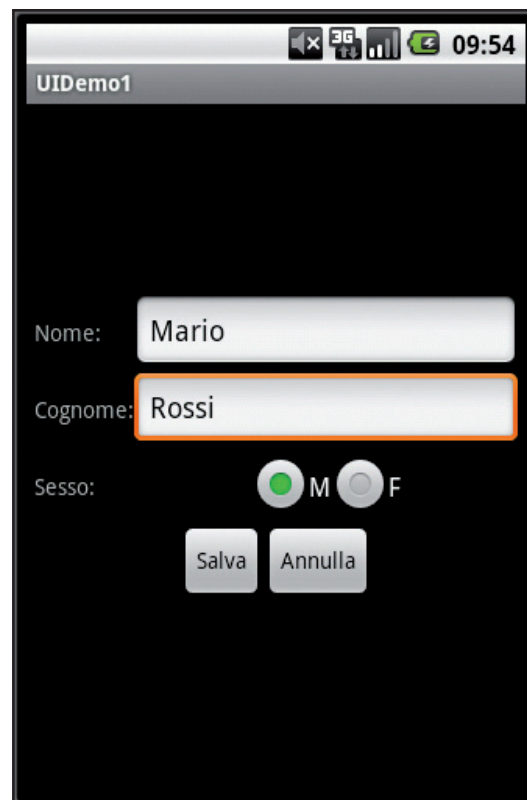



Fig. 1: Un form che permette all'utente l'inserimento dei suoi dati anagrafici di base

ANDROID 2.0.1 (API 6)

Anche questo mese c'è un aggiornamento di Android da segnalare (sono velocissimi e spesso quando la rivista è già in edicola già è disponibile l'aggiornamento successivo!). Ai primi di Dicembre è stata pubblicata la versione 2.0.1 del sistema, che corrisponde alle API livello 6 (la versione precedente era la 2.0 - API 5). In questo caso si tratta di un rilascio di manutenzione, che corregge qualche bug ed introduce solo piccoli cambi nelle API, perlopiù relativi allo strato Bluetooth e alla nuova gestione dei contatti introdotta con la versione 2.0. Si segnala poi che anche l'SDK è stato aggiornato alla versione r4. È possibile aggiornare i propri componenti servendosi della procedura automatizzata descritta nel numero scorso, oppure scaricando il nuovo software a partire dall'indirizzo: http://developer.android.com/sdk/

```
public class XMLLayoutDemo1Activity extends Activity
{

        @Override
        public void onCreate(Bundle
                                savedInstanceState)
{
super.onCreate(savedInstanceState);
                setContentView(R.layout.main);
        }

}
```

Sul CD-Rom allegato alla rivista trovate il progetto Android completo avviabile con l'emulatore.

XML
Se non siete pratici di XML e di linguaggi a marcatori correte il rischio di non condividere quanto ribadito in questo articolo, e di considerare più semplice il design delle interfacce utente a mezzo di un linguaggio di programmazione come Java. Se la pensate così, vi garantisco, è solo perché ancora non siete riusciti a cogliere l'essenza di XML e del suo utilizzo. Vi anticipo che non dovete farvi spaventare: XML è nato per essere semplice, e dunque è anche facile. Ecco alcuni link per studiare cosa sia XML e come funzioni:

http://it.wikipedia.org/wiki/XML
http://xml.html.it/guide/leggi/58/guida-xml-di-base/
http://www.mrwebmaster.it/xml/guide/guida-xml_9/
http://www.risorse.net/xml/guida.asp

REGOLE GENERALI

Entriamo un po' più nel dettaglio del formato XML usato per descrivere un layout. Come è facile intuire, i tag adoperabili al suo interno corrispondono ai nomi dei widget e dei layout manager descritti nell'articolo del mese scorso. In pratica al widget Java *android.widget.TextView* corrisponde il tag XML *<TextView>*, al layout manager *android.widget.LinearLayout* corrisponde *<LinearLayout>*, e così via. Potete riprendere il numero del mese scorso e desumere da voi le corrispondenze fra classi e tag. È invece importante sapere che il namespace da utilizzare è: http://schemas.android.com/apk/res/android Normalmente lo si fa dichiarando il namespace nel primo tag utilizzato (quello di ordine superiore), abbinandogli lo shortname android. In breve, il modello da seguire è il seguente:

```
<tag1 xmlns:android="http://
                schemas.android.com/apk/res/android"
        android:attr1="val1"
        android:attr2="val2">
  <tag2 android:attr1="val1" android:attr2="val2" />
  <tag2 android:attr1="val1" android:attr2="val2" />
</tag1>
```

Là dove gli attributi sono utilizzati per immettere Un valore libero, ad esempio un testo, una dimensione, un colore, un'immagine e così via, è possibile sfruttare i riferimenti ad altri materiali conservati nella gerarchia della cartella *res*. Quando è stato trattato lo speciale file *AndroidManifest.xml* (cfr. ioProgrammo 144) abbiamo imparato ad usare i riferimenti del tipo:

```
@tipo/nome
```

Ad esempio, per richiamare la stringa "titoloApplicazione" definita in un XML sotto *res/values*, è possibile fare:

```
@string/titoloApplicazione
```

Bene, la stessa identica cosa può essere fatta in un XML di layout. Ad esempio, si può ricorrere a questa funzionalità quando si imposta il testo mostrato in un bottone. Invece di scrivere:

```
<Button android:text="Salva" />
```

Si può scrivere:

```
<Button
     android:text="@string/etichettaBottoneSalva" />
```

A patto, ovviamente, di aver creato in *res/values* un file XML che definisca la risorsa stringa *etichettaBottoneSalva*, come ad esempio:

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
            <string
        name="etichettaBottoneSalva">Salva</string>
</resources>
```

La regola vale non soltanto per le stringhe, ma per tutte le categorie di risorse. Ricapitoliamole:

- **@array**, per gli array.
- **@color**, per i colori.
- **@dimen**, per le dimensioni.
- **@drawable**, per i valori *drawable*, ma anche per le immagini messe in *res/drawable*.
- **@layout**, per richiamare altri layout.
- **@raw**, per i file nella cartella *res/raw*.
- **@string**, per le stringhe.
- **@style**, per gli stili.

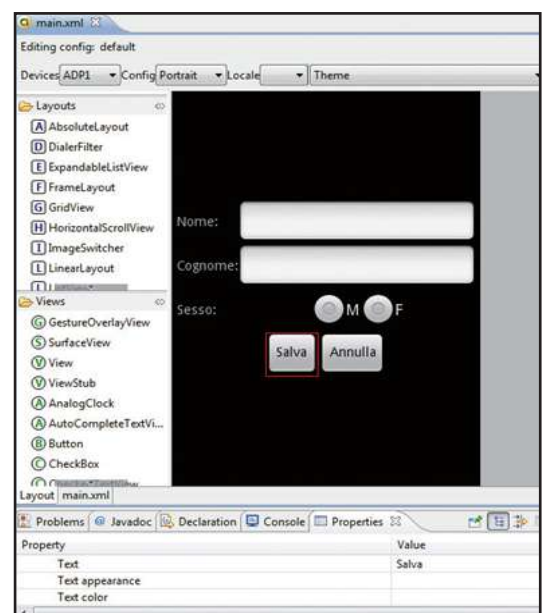



Fig. 2: L'editor visuale compreso con ADT per la creazione guidata dei layout XML in ambiente Eclipse

ASSEGNARE UN ID AI COMPONENTI

A ciascun componente dichiarato nell'XML, sia esso un contenitore o un widget, è possibile assegnare un identificativo, utile per rintracciare successivamente il componente. Per assegnare un identificativo si deve utilizzare l'attributo *id*:

```
<Tag android:id="..." />
```

Per assegnare l'ID è necessario seguire una particolare sintassi, basata sul seguente modello:

```
@+nomeGruppo/nomeId
```

Questa sintassi fa sì che nella classe *R* venga introdotto, se non esiste già, il gruppo *nomeGruppo*, e che al suo interno venga memorizzato il riferimento all'ID *nomeId*. Sarà pertanto possibile ricorrere all'ID, nel codice Java, usando il riferimento:

```
R.nomeGruppo.nomeId
```

Facciamo il caso di questo bottone:

```
<Button android:id="@+idBottoni/salva"
                    android:text="Salva" />
```

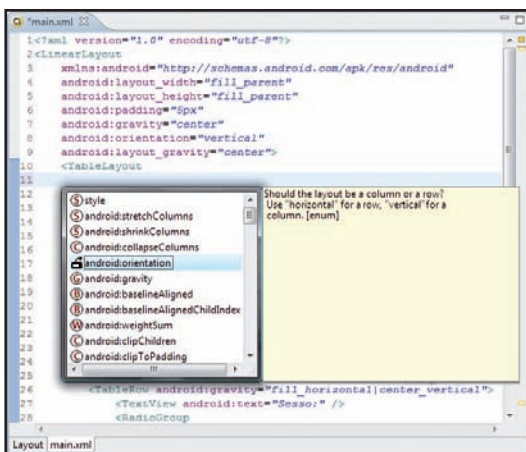



Fig. 3: Anche passando all'editing manuale dei layout XML è possibile usufruire di alcune agevolazioni in ambiente Eclipse ADT, come l'auto-completamento e l'help in linea per tag ed attributi

In Java sarà possibile richiamarlo adoperando il riferimento:

```
R.idBottoni.salva
```

Ad esempio le attività dispongono del metodo *findViewById()* che, come il nome lascia presupporre, ricerca nel layout caricato un componente avente l'ID specificato come argomento. Il bottone può a questo punto essere recuperato e manipolato (ad esempio per aggiungere un gestore di evento, come impareremo prossimamente):

```
Button button = (Button)
          findViewById(R.idBottoni.salva);
```

ATTRIBUTI COMUNI

Gli attributi applicabili ad un tag variano a seconda del componente cui fanno riferimento. In alcuni casi gli attributi sono obbligatori, mentre in altri sono opzionali. Per esplorare tutti gli attributi di ogni specifico widget, di conseguenza, è meglio affidarsi alle procedure guidate di un editor visuale, come quello di Eclipse ADT descritto sopra. Gli stessi attributi, naturalmente, sono esaustivamente riportati anche nella documentazione ufficiale. Tutto ciò per dire che, in generale, non ci inalbereremo subito nello studio di tutti gli attributi possibili.
Ci concentreremo invece su quelli comuni o comunque più interessanti. I primi due che esaminiamo sono sempre obbligatori, e si chiamano *layout_width* e *layout_height*. Servono per decretare come si relazioni l'oggetto rispetto alle dimensioni del suo contenitore. Due sono i valori possibili:

- **wrap_content**
  Rende il componente grande tanto quanto impongono i suoi sotto-componenti. In pratica tutti i sotto-componenti vengono dimensionati rispetto, possibilmente, alla loro dimensione ideale, e poi il contenitore che li contiene viene dimensionato di conseguenza.
- **fill_parent**
  Allarga il componente fino a fargli occupare tutto lo spazio a sua disposizione concessogli dal suo contenitore d'ordine superiore.

Ad esempio:

```
<Button android:layout_width="wrap_content"
          android:layout_height="wrap_content"
                    android:text="Salva" />
```

Questo bottone sarà grande tanto quanto basta a mostrare la scritta "*Salva*". Quest'altro invece:

```
<Button android:layout_width="fill_parent"
            android:layout_height="fill_parent"
                    android:text="Salva" />
```

Sarà allargato, sia in orizzontale sia in verticale, fino ad occupare tutto lo spazio messo a sua disposizione. Altri attributi condivisi da tutti i componenti sono quelli definiti nella classe *View*,

NAMESPACE

I *namespace* di XML, per semplificare, sono l'equivalente dei package in Java. Chi "inventa" un nuovo gruppo di tag, per regola, deve inserirli in un namespace, in modo che possano essere utilizzati senza andare in collisione con eventuali altri tag omonimi di un altro autore. I package di Java hanno la forma *aaa.bbb.ccc*, mentre i namespace di XML sono un URL del tipo http://aaa/bbb/ccc. Benché questo sia un indirizzo Web a tutti gli effetti, non è detto che debba per forza esserci un file da scaricare a quel percorso, anche se in genere l'autore ci mette la definizione dei tag e degli attributi previsti dal namespace. A ciascuno dei namespace importati in un tag XML è possibile associare uno *shortname*, cioè un nome breve, utile per richiamare tag ed attributi del namespace senza ambiguità. Per approfondire i namespace e per qualche esempio pratico:

http://xml.html.it/articoli/leggi/1648/il-misterioso-mondo-dei-namespaces/

ereditati poi da tutti i widget e tutti i layout. Fra questi si segnalano:

- **minWidth e minHeight**
  Permettono di specificare una dimensione minima per il componente. La misura deve essere espressa letteralmente, specificando l'unità di misura, oppure riferendo una dimensione definita sotto *res/values*. (cfr. ioProgrammo 144).

- **paddingLeft, paddingTop, paddingRight, paddingBottom** e **padding**
  Permettono di specificare il padding (margine interno) del componente (cfr. ioProgrammo 146). I primi quattro attributi permettono di distinguere la misura di padding assegnata in ogni direzione, mentre il quinto permette di assegnare a tutte e quattro le direzioni il medesimo padding con un'istruzione sola. I valori espressi devono essere dotati di unità di misura, o in alternativa fare riferimento ad una dimensione nota in *res/values*.

- **visibility**
  Accetta tre possibili valori: 0 (visibile), 1 (invisibile), 2 (scomparso). Nel primo caso, che è quello di default, il componente è visibile. Nel secondo caso il componente non è visibile, ma lo spazio spettante viene riservato e mantenuto vuoto. Nel terzo caso il componente è invisibile e nessuno spazio gli viene assegnato durante il disegno del layout, proprio come non fosse mai esistito. La visibilità di un componente può essere poi manipolata a runtime da codice, con il metodo *setVisibility()*.

Esistono poi dei tag non comuni a tutti i widget, ma molto diffusi. Tra questi:

- **width e height**
  Permettono di stabilire una dimensione precisa del componente, attraverso una misura scritta letteralmente o riferita da *res/values*.

- **maxWidth e maxHeight**
  Permettono di stabilire una dimensione massima del componente, attraverso una misura scritta letteralmente o riferita da *res/values*.

- **gravity**
  Stabilisce la gravità applicata al componente. I valori possibili sono: *top, bottom, left, right, center_vertical, center_horizontal, center, fill_horizontal, fill_vertical, fill, clip_vertical* e *clip_horizontal*.

I tanti componenti che derivano da *TextView* (tra cui i diversi tipi di bottoni) hanno a loro disposizione:

- **text**
  Permette di impostare il testo visualizzato, attraverso un valore letterale o un riferimento a stringa memorizzata sotto *res/values*.

Una particolare estensione di *TextView* è *EditText*. Con *EditText* si realizza una casella di input, che permette all'utente di immettere del testo. In questo caso possono tornare utili i seguenti attributi:

- **hint**
  Un suggerimento da visualizzare quando non c'è del testo visualizzato.

- **password**
  Un booleano (*true* o *false*) che indica se il campo contiene una password. In questo caso il testo viene camuffato in modo da non risultare leggibile ad un occhio indiscreto.

- **numeric**
  Rende il campo di tipo numerico. Si deve specificare uno o più valori (separandoli con *pipe*) fra *integer*, *signed* e *decimal*. Il primo indica un valore intero, il secondo un valore numerico con segno, il terzo di un valore decimale.

- **digits**
  Da usare se il campo è numerico. Permette di specificare quali cifre è possibile utilizzare. Ad esempio il valore "*123*" farà sì che l'utente possa inserire nel campo sono le cifre "*1*", "*2*" e "*3*".

Vi invitiamo a consultare la documentazione ufficiale, per completare da voi la panoramica facendo qualche esperimento anche con gli attributi meno ricorrenti.

Fig. 4: Dimensionare un componente rispetto al suo contenitore. Da sinistra a destra: wrap_content su layout_width e layout_height; wrap_content su layout_width e fill_parent su layout_height; fill_parent su layout_width e wrap_content su layout_height; fill_parent su layout_width e su layout_height.

PROSSIMAMENTE

Il prossimo mese proseguiremo la nostra trattazione delle interfacce utente, parlando di eventi.

Carlo Pelliccia

L'AUTORE

Carlo Pelliccia lavora presso 4IT (www.4it.it), dove si occupa di analisi e sviluppo software per piattaforme Java. Nella sua carriera di technical writer ha pubblicato cinque manuali ed oltre centocinquanta articoli, molti dei quali proprio tra le pagine di ioProgrammo. Il suo sito, che ospita anche diversi progetti Java Open Source, è disponibile all'indirizzo www.sauronsoftware.it

GESTIRE IL TOUCH SU ANDROID

SESTO APPUNTAMENTO. IN QUESTA PUNTATA DEL CORSO IMPAREREMO LE VARIE TECNICHE PER INTERCETTARE LE AZIONI DI TOCCO E DIGITAZIONE ESEGUITE DALL'UTENTE SUI WIDGET PRESENTI NEL DISPLAY, IN MODO DA REAGIRE DI CONSEGUENZA

Nei due precedenti appuntamenti abbiamo conosciuto i principali widget di Android e le tecniche utili per richiamarli e disporli nel display dello smartphone. Oggi impareremo a raccogliere l'input dell'utente, e lo faremo intercettando gli eventi scatenati dai widget.

I METODI DI CALLBACK

Tutti i widget di Android dispongono di una serie di metodi di callback, con nomi del tipo *onTipo Evento()*. Questi metodi vengono richiamati automaticamente ogni volta che il corrispondente evento è riscontrato sul widget. Mi spiego meglio con un esempio concreto. La classe *android.widget.Button*, che abbiamo conosciuto nel corso delle due puntate precedenti, definisce uno speciale metodo chiamato *onTouchEvent()*. Questo metodo è eseguito automaticamente ogni volta che il bottone viene toccato dall'utente, attraverso il touch screen del suo dispositivo. Sono molti i metodi di questa categoria, ed ogni widget ha i propri. Ciascun metodo ha le sue regole e la sua firma: parametri e valore di ritorno, insomma, cambiano di caso in caso. La documentazione ufficiale delle API di Android, come sempre, presenta l'elenco esaustivo (in inglese) per ciascun widget. Lo sviluppatore interessato alla gestione di uno specifico evento deve individuare il metodo di suo interesse, quindi estendere il widget e ridefinire il metodo individuato. Proviamo a farlo proprio con il metodo *onTouchEvent()* di *Button*, rendendo il bottone reattivo al singolo "clic da dito":

```
package it.ioprogrammo.buttonclickdemo1;

import android.content.Context;
import android.view.MotionEvent;
import android.widget.Button;
import android.widget.Toast;

public class MyButton extends Button {
    public MyButton(Context context) {
```

```
        super(context);
    }

    @Override
        public boolean onTouchEvent(MotionEvent event) {
            super.onTouchEvent(event);
            int action = event.getAction();
            if (action == MotionEvent.ACTION_DOWN) {
                Toast toast = Toast.makeText(getContext(),
                                            "Bottone cliccato!",
                                            Toast.LENGTH_SHORT);
                toast.show();
                return true;
            }
        return false;
    }
}
```

Il metodo *onTouchEvent()* riceve in ingresso un argomento di tipo *android.view.MotionEvent*, che riporta tutte le informazioni relative all'evento di tocco riscontrato sul bottone. La versione superiore del metodo (cioè quella definita in *Button*) viene richiamata con la riga:

```
super.onTouchEvent(event);
```

Questo, nel caso specifico, è necessario affinché l'evento venga gestito graficamente: la definizione base del metodo contenuta in *Button* fa sì che il bottone, quando sottoposto a pressione, cambi il suo aspetto ed il suo colore, per mostrare reattività al tocco dell'utente (con il tema predefinito di Android 2.0, il bottone cambia da grigio ad arancione).
Dopo aver eseguito la versione base del metodo, la nostra versione ridefinita controlla i dettagli dell'evento. Se si tratta del primo tocco riscontrato sul bottone (*ACTION_DOWN*), viene mostrato all'utente un messaggio di notifica ("Bottone cliccato!"). Altri filtri possono essere applicati per intercettare ulteriori varianti dell'evento di tocco, ad esempio *ACTION_MOVE* per un trascinamento e *ACTION_UP* per l'azione di rilascio del pulsante

(dito sollevato dal display).
Il metodo *onTouchEvent()*, infine, richiede la restituzione di un valore booleano, che serve per indicare se l'evento è stato effettivamente gestito oppure no. L'implementazione restituisce *true* nel caso di un evento *ACTION_DOWN*, effettivamente gestito attraverso il messaggio mostrato all'utente, mentre restituisce false in tutti gli altri casi.

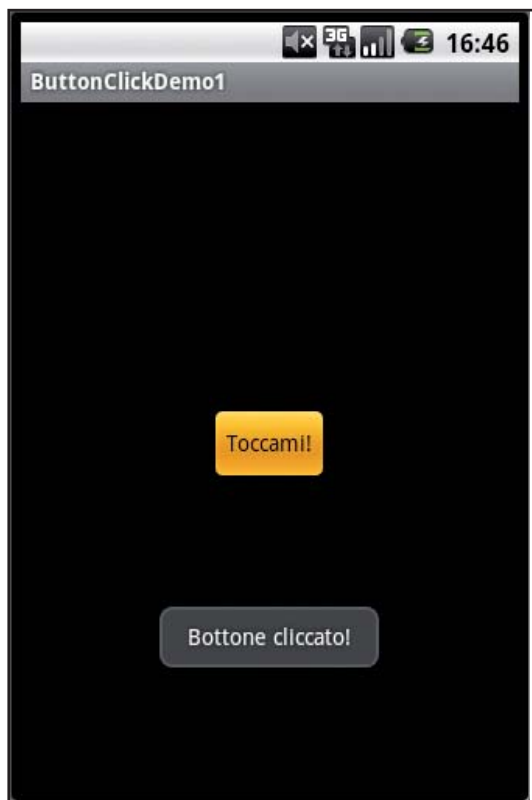



Fig. 1: In questo esempio l'evento di primo tocco sul bottone viene gestito mostrando un avviso all'utente

La classe *MyButton* può essere ora impiegata all'interno di un'attività, come abbiamo imparato a fare due numeri fa:

```
package it.ioprogrammo.buttonclickdemo1;

import android.app.Activity;
import android.os.Bundle;
import android.view.Gravity;
import android.widget.LinearLayout;

public class ButtonClickDemo1Activity extends Activity {
    @Override
        public void onCreate(Bundle savedInstanceState) {
          super.onCreate(savedInstanceState);
          MyButton button = new MyButton(this);
          button.setText("Toccami!");
          LinearLayout layout = new LinearLayout(this);
          layout.setGravity(Gravity.CENTER);
          layout.addView(button);
```

```
          setContentView(layout);
     }
}
```

EVENT LISTENER

La ridefinizione dei metodi di callback è una tecnica che funziona, ma non è molto pratica: ogni volta che si usa un widget bisogna estenderlo e generare quindi un'altra classe, in modo da poter ridefinire il metodo (o i metodi) di callback di proprio interesse. Questo, di per sé, è scomodo e prolisso. Quanti widget possono esserci in un'applicazione di media complessità? Fra bottoni, checkbox, etichette, campi di testo e via discorrendo, il numero è sicuramente elevato. Ecco allora che la velocità di sviluppo viene frenata dal dover definire e catalogare tante classi quanti sono i widget che si vogliono utilizzare. Un vero incubo! La ridefinizione dei metodi di callback, pertanto, è una pratica che serve solo in determinati casi, principalmente durante la creazione di componenti custom. Ad esempio è possibile estendere *View* per creare un componente personalizzato. È lecito, lo si può fare, e probabilmente prima o poi vi servirà di farlo. Per un intervento di questo genere, dunque, non c'è nulla di male nel ridefinire ed utilizzare i metodi di callback, anzi l'operazione sarebbe sicuramente necessaria. Per tutti gli altri usi quotidiani dei widget pronti all'uso, invece, Android mette a disposizione un meccanismo più semplice e sbrigativo, basato sull'utilizzo dei cosiddetti *event listener*. Scopriamo insieme di cosa si tratta. Tutti i widget mettono a disposizione una seconda serie di metodi, questa volta del tipo *setOnTipo EventoListener()*. Il widget *Button*, ad esempio, dispone del metodo *setOnClickListener()*. Attraverso i metodi di questa categoria è possibile registrare al widget degli *event listener*, cioè delle istanze di speciali classi, realizzate appositamente per ricevere notifica ogni volta che lo specifico evento accade. Per ciascun differente tipo di event listener esiste un'interfaccia apposita, che lo sviluppatore deve implementare per creare il suo gestore dell'evento. Ad esempio, l'interfaccia da implementare per gli eventi di clic è *android.view. View.OnClickListener* (interfaccia innestata nella classe *View*). Ciascuna interfaccia, ovviamente, richiede l'implementazione di uno o più metodi. Nel caso di *OnClickListener*, per proseguire con l'esempio, il metodo da ridefinire è:

```
public void onClick(View v) { ... }
```

Proviamo a ripetere l'esempio del paragrafo precedente, questa volta utilizzando il principio degli event listener e, più nello specifico, l'interfaccia *OnClickListener*:

TOAST
"*Toast*" è la parola usata nel gergo dei dispositivi mobili per identificare le finestrelle pop-up con un avviso rivolto all'utente. Si chiamano così perché nelle loro implementazioni più classiche spuntano fuori dalla parte bassa del display e somigliano ad un toast che salta fuori dal tostapane a cottura ultimata. In Android, come si evince dagli esempi mostrati nell'articolo, gli avvisi toast possono essere realizzati servendosi della classe *android.widget.Toast*. Ad ogni modo, ne parleremo nuovamente in seguito.

FOCUS

Un altro evento comune definito in *View* è quello relativo al *focus*. Quando un widget riceve il focus, significa che è selezionato, e che tutti gli eventi di digitazione saranno a esso rivolti. È possibile sapere quando un widget riceve o perde il focus. Il metodo per registrare il listener sul widget è *setOnFocusChangeListener()*, mentre l'interfaccia per implementarlo è *View.OnFocusChangeListener*. L'interfaccia richiede il metodo *onFocusChange(View view, boolean hasFocus)*. Il parametro view è il widget che ha subito l'evento, mentre il booleano *hasFocus* indica se il componente ha ricevuto il focus (*true*) oppure se lo ha perso (*false*).

```
package it.ioprogrammo.buttonclickdemo2;

import android.view.View;
import android.view.View.OnClickListener;
import android.widget.Toast;

public class MyClickListener implements OnClickListener {
...
```

Questa volta, invece di estendere *Button* e realizzare così un componente custom, abbiamo semplicemente implementato un'interfaccia. Nel metodo *onClick()* abbiamo scritto il codice necessario per gestire l'evento di clic. Il parametro ricevuto dal metodo, nel caso specifico, rappresenta l'oggetto *View* o derivato sul quale l'evento è stato riscontrato. Affinché la classe *MyClickListener* venga utilizzata come gestore dell'evento di clic su uno specifico widget, è necessario registrarne un'istanza sul widget stesso, servendosi del metodo *setOnClickListener()* citato in precedenza. Lo si può fare quando si allestisce o si richiama il layout dalla schermata. Ecco una *Activity* equivalente a quella del paragrafo precedente, ma che a differenza di quest'ultima utilizza un widget *Button* standard insieme con l'event listener di poco sopra:

```
package it.ioprogrammo.buttonclickdemo2;

import android.app.Activity;
import android.os.Bundle;
import android.view.Gravity;
import android.widget.Button;
import android.widget.LinearLayout;

public class ButtonClickDemo2Activity extends Activity {
...
```

In questa maniera non è stato necessario creare un componente custom: è stato sufficiente registrare sul *Button* l'event listener realizzato pocanzi, con la riga:

```
button.setOnClickListener(new MyClickListener());
```

La tattica degli event listener, inoltre, si sposa meglio con la possibilità messa in campo da Android di definire risorse e layout attraverso dei file XML (cfr. numero precedente). Il layout realizzato nell'attività mostrato poco sopra, ad esempio, potrebbe essere definito in un file XML indipendente. Chiamiamolo *main.xml*:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.
                    android.com/apk/res/android"
    android:orientation="vertical"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    android:gravity="center">
    <Button android:id="@+id/bottone01"
     android:layout_width="wrap_content"
     android:layout_height="wrap_content"
     android:text="Toccami!" />
</LinearLayout>
```

L'attività, di conseguenza, andrebbe riscritta alla seguente maniera:

```
package it.ioprogrammo.buttonclickdemo3;

import android.app.Activity;
import android.os.Bundle;
import android.widget.Button;
public class ButtonClickDemo3Activity extends Activity {
...
```

Dopo aver richiamato il layout definito nel file XML, non si deve far altro che recuperare il bottone al quale si vuole collegare l'evento e registrare su di esso il proprio listener personalizzato.

COME SCRIVERE MENO CODICE

Qualcuno potrebbe obiettare che, con gli event listener, è comunque necessario creare una classe distinta per ciascun gestore previsto, con il rischio di avere più codice dedicato alla cattura degli eventi che non alla loro gestione. Esistono diversi trucchi applicabili con gli event listener che aiutano ad evitare le situazioni di questo tipo. Ve ne svelo un paio. Per realizzare un event listener bisogna estendere un'interfaccia. Java non supporta l'ereditarietà multipla, e quindi una classe può avere una sola super-classe. Questo limite però non vale nel caso delle interfacce: una classe ne può implementare un numero qualsiasi. Ecco allora che, nel caso di GUI non troppo complesse, si può fare che la *Activity* che controlla lo schermo sia essa stessa event listener di uno o più eventi, per uno o più widget. Mi spiego meglio con un esempio. Prendiamo in considerazione il seguente layout, come al solito da battezzare *main.xml*:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas
                    .android.com/apk/res/android"
    android:orientation="vertical"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    android:gravity="center">
    <Button android:id="@+id/bottone01"
```

```
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="Bottone 1" />
    <Button android:id="@+id/bottone02"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="Bottone 2" />
</LinearLayout>
```

L'esempio è lievemente più complesso del precedente: qui i bottoni sono diventati due.
Realizziamo un'attività che carichi questo layout e gestisca gli eventi di clic su ambo i bottoni:

```
package it.ioprogrammo.buttonclickdemo4;

import android.app.Activity;
import android.os.Bundle;
import android.view.View;
import android.view.View.OnClickListener;
import android.widget.Button;
import android.widget.Toast;

public class ButtonClickDemo4Activity extends Activity
                                    implements OnClickListener {
...
```

In questo caso è la stessa attività a implementare l'interfaccia *OnClickListener*, definendo di conseguenza il metodo *onClick()*. Non è stato dunque necessario creare una classe apposita. Inoltre l'event listener realizzato è stato adoperato su due bottoni differenti (*bottone01* e *bottone02*). È stato possibile farlo servendosi degli id assegnati ai due bottoni: all'interno del codice del solo metodo *onClick()* realizzato, si è mostrato un messaggio differente a seconda della sorgente dell'evento. Un'altra tecnica per risparmiare codice consiste nell'adoperare le classi innestate anonime di Java. È possibile fare qualcosa di questo tipo:

```
button.setOnClickListener(new OnClickListener() {
    @Override
    public void onClick(View view) {
         // gestione evento
    }
});
```

Di fatto si crea e si registra allo stesso tempo il gestore dell'evento di clic. Ci pensa il compilatore a separare la classe anonima innestata su un file *.class* differente. Riscriviamo allora l'esempio precedente secondo quest'altra tecnica:

```
package it.ioprogrammo.buttonclickdemo5;

import android.app.Activity;
```

```
import android.os.Bundle;
import android.view.View;
import android.view.View.OnClickListener;
import android.widget.Button;
import android.widget.Toast;

public class ButtonClickDemo5Activity extends Activity {
...
```

Così facendo non c'è bisogno di far implementare alcuna interfaccia all'attività. La tecnica consente di scrivere davvero poco codice per intercettare gli eventi, lasciando il programmatore libero di concentrarsi sulla logica della loro gestione. Il più delle volte, è proprio quest'ultima la tecnica da preferirsi.

PANORAMICA SUGLI EVENTI

Ora che abbiamo imparato ad utilizzare gli event listener, la prima cosa da fare è chiedersi: quanti e quali sono gli eventi che è possibile gestire? La risposta non è semplice: ogni widget ha i suoi eventi e, di conseguenza, i suoi event listener. Anche in questo caso, quindi, la documentazione è una risorsa fondamentale, che bisogna assolutamente imparare a leggere. Soprattutto nel caso dei widget più particolari, quindi, dovrete cavarvela, altrimenti rischiamo di prolungare questo corso fino al 2020! Insieme, però, possiamo prendere in esame i tipi di eventi più universali e diffusi, riconosciuti e gestibili su qualsiasi widget. A definirli, tanto per cambiare, è la madre di tutti i widget: la classe *android.view.View*. Ecco una panoramica degli eventi più importanti:

- **Click**. Lo abbiamo usato in tutti gli esempi precedenti. Il metodo sul widget è *setOnClickListener()*, e l'interfaccia per il gestore da implementare è *View.OnClickListener*. Il metodo richiesto dall'interfaccia è *onClick(View view)*, che in parametro riporta il widget che ha subito l'evento.

- **Click lungo**. Un evento che accade quando si clicca su un widget e si mantiene la pressione per qualche istante. Il metodo per registrare l'event listener è *setOnLongClickListener()*, e l'interfaccia per il gestore è *View.OnLongClickListener*. Il metodo da implementare è *onLongClick(View view)*. Il metodo, come nel caso precedente, riceve come parametro un riferimento al widget su cui si è prodotto l'evento. In più, il metodo deve restituire un booleano per segnalare se l'evento è stato completamente gestito (*true*) oppure no (*false*).

MODIFICATORI

Quando si riceve un evento di digitazione, attraverso l'istanza di *KeyEvent*, è possibile sapere se insieme al tasto principale che riguarda l'evento sono stati attivati anche uno o più tasti modificatori. I tasti modificatori, in Android, sono tre: ALT, SHIFT e SYM. *KeyEvent* permette di controllare lo stato dei tre modificatori attraverso i metodi booleani *isAltPressed()*, *isShiftPressed()* e *isSymPressed()*.

Fig. 2: Questa volta bisogna gestire i clic su due bottoni differenti

- **Tocco**. Un evento più generico dei due precedenti: serve per rilevare un tocco qualsiasi su un componente. Il metodo per registrare il listener sul widget è *setOnTouchListener()*, mentre l'interfaccia per implementarlo è *View.OnTouch Listener*. L'interfaccia richiede il metodo *onTouch(View view, MotionEvent event)*. Come nei casi precedenti, view è il widget che ha subito l'evento. Il parametro di tipo *MotionEvent* riporta invece i dettagli dell'evento (tipo di azione, coordinate, durata ecc.), come nel caso documentato nel primo esempio di oggi. Il metodo deve restituire un booleano per segnalare se l'evento è stato completamente gestito (*true*) oppure no (*false*).

- **Digitazione**. Un evento usato per segnalare la pressione o il rilascio di un tasto della tastiera hardware. Il metodo per registrare il listener sul widget è *setOnKeyListener()*, mentre l'interfaccia per implementarlo è *View.OnKeyListener*. L'interfaccia richiede il metodo *onKey(View view, int keyCode, KeyEvent event)*. Come nei casi precedenti, view è il widget che ha subito l'evento. Il parametro *keyCode* riporta il codice associato al tasto premuto, mentre quello di tipo *KeyEvent* riporta ulteriori dettagli (tasto pigiato, tasto rilasciato, eventuali modificatori e così via). Il metodo deve restituire un booleano per segnalare se l'evento è stato completamente gestito (*true*) oppure no (*false*).

Siccome di esempi dedicati ai clic ne abbiamo già visti diversi, proviamo ora a gestire i due eventi di tocco e digitazione. Prepariamo il seguente layout, da chiamare come al solito *main.xml*:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.
                    android.com/apk/res/android"
   android:orientation="vertical"
   android:layout_width="fill_parent"
   android:layout_height="fill_parent"
   android:gravity="center">
   <EditText android:id="@+id/eventArea"
   android:layout_width="wrap_content"
   android:layout_height="wrap_content"
   android:background="#990000"
   android:textColor="#FFFFFF"
   android:width="200px"
   android:height="200px" />
   <TextView android:id="@+id/logArea"
   android:layout_width="wrap_content"
   android:layout_height="wrap_content"
   android:text="Compi un evento sull'area
                    rossa qui sopra" />
</LinearLayout>
```

***Fig. 3:** L'applicazione intercetta gli eventi di digitazione e tocco riscontrati sull'area rossa, e poi li descrive all'utente che li ha compiuti*

Il layout dispone, al centro del display, due componenti. C'è un *EditText* con sfondo di colore rosso e dimensioni 200 x 200 pixel, che useremo per far compiere all'utente delle operazioni di digitazione e di tocco. C'è poi un *TextView*, che useremo invece per descrivere ogni evento riscontrato sul componente precedente, dimostrando così di averlo intercettato correttamente.
Andiamo ora a realizzare, mediante una *Activity*, quanto proposto:

```
package it.ioprogrammo.eventdemo;

import android.app.Activity;
import android.os.Bundle;
import android.util.Log;
import android.view.KeyEvent;
import android.view.MotionEvent;
import android.view.View;
import android.view.View.OnTouchListener;
import android.view.View.OnKeyListener;
import android.widget.TextView;

public class EventDemoActivity extends Activity {
...
```

Sull'*EditText* vengono registrati due listener, uno per il tocco e l'altro per la digitazione. La tecnica usata per gli event listener è quella della classe anonima innestata. Gli eventi intercettati vengono esaminati (attraverso le istanze di *MotionEvent* e *KeyEvent*) e descritte, sia nei log dell'applicazione (che potrete controllare comodamente usando Eclipse) sia nella *TextView* che ne dà immediato riscontro all'utente.

Carlo Pelliccia

Carlo Pelliccia lavora presso 4IT (www.4it.it), dove si occupa di analisi e sviluppo software per piattaforme Java. Nella sua carriera di technical writer ha pubblicato cinque manuali ed oltre centocinquanta articoli, molti dei quali proprio tra le pagine di ioProgrammo. Il suo sito, che ospita anche diversi progetti Java Open Source, è disponibile all'indirizzo www.sauronsoftware.it

ANDROID: TUTTO SUI MENU

SETTIMO APPUNTAMENTO. ARGOMENTO DEL MESE SONO I MENU. LE APPLICAZIONI ANDROID NE SUPPORTANO DIVERSI TIPI, CHE L'UTENTE PUÒ SFRUTTARE PER AZIONARE COMANDI E IMPOSTARE LE OPZIONI. CONOSCIAMOLI E IMPARIAMO A PROGRAMMARLI

I menu sono una parte importante di qualsiasi applicazione. Gli utenti sono abituati ad avere a che fare con il concetto di menu, al quale si rivolgono ogni volta che vogliono cercare i comandi o modificare le opzioni delle loro applicazioni. Ciò risulta vero tanto nell'ambito dei software desktop, quanto in quello delle applicazioni mobili per cellulari e smartphone. Con questo articolo impareremo a conoscere i menu previsti da Android, ponendo particolare attenzione alle regole per la costruzione di interfacce grafiche semplici ed efficaci, con menu concisi e facili da raggiungere e navigare.

I MENU IN ANDROID

In Android esistono tre differenti tipi di menu, che lo sviluppatore può collegare ad una Activity:

- ***Options menu***
  Sono i menu concepiti per raggruppare le opzioni ed i comandi di un'applicazione. Si dividono in due sotto-gruppi, *icon menu* ed *expanded menu*, descritti di seguito.

  - ***Icon menu***
    Sono i menu con le opzioni principali di un'applicazione. Vengono visualizzati nella parte bassa dello schermo quando si schiaccia il tasto "menu" del dispositivo. Vengono chiamati *icon menu* perché gli elementi contenuti al loro interno, in genere, sono delle grosse icone che l'utente può selezionare con il polpastrello. Costituiscono il menu principale di ogni attività e dovrebbero contenere sempre e solo le opzioni più importanti. Questi menu sono di rapido accesso, ma soffrono per questo di alcune limitazioni: possono contenere al massimo sei elementi, e non è possibile inserire negli icon menu elementi avanzati come le caselle di spunta (*checkbox*) e i bottoni radio.

  - ***Expanded menu***
    Quando il primo tipo di menu non è sufficiente per esporre tutti i comandi e tutte le opzioni di un'applicazione, le attività fanno ricorso agli *expanded menu* (letteralmente *menu espansi*). Quando ciò avviene, il menu principale, come suo ultimo tasto, presenta il bottone "*altro*". Attivandolo si accede ad una lista aperta a tutto schermo, che permette la consultazione delle altre opzioni di menu.

Fig. 1: L'icon menu utilizzato dal browser di Android

- ***Context menu***
  I menu contestuali sono quelli che appaiono quando si mantiene il tocco per qualche istante su un widget che ne è dotato. Ad esempio nel browser è possibile eseguire un tocco di questo tipo sopra ad un'immagine. Dopo qualche istante verrà aperto un menu contestuale con alcune opzioni relative alla pagina corrente e all'immagi-

ne selezionata, come ad esempio i comandi per salvarla in locale e condividerla con gli amici. Come nel caso precedente, questo genere di menu si presenta come una lista a tutto schermo, che può contenere numerose opzioni.

- ***Submenu***
  Le applicazioni che dispongono di molti comandi possono usufruire anche dei submenu. In pratica, in uno qualsiasi dei menu descritti in precedenza, è possibile inserire un elemento che, invece di compiere un'azione diretta, va ad aprire un sotto-menu, nel quale si possono presentare ulteriori possibilità di scelta.

Nei prossimi paragrafi impareremo a programmare tutti e tre i tipi di menu presentati.

OPTIONS MENU

Cominciamo dagli options menu che, come abbiamo detto, costituiscono il menu principale di qualsiasi applicazione. Il menu delle opzioni è un concetto strettamente legato a quello di singola attività. Ogni *Activity*, infatti, può avere un solo options menu. La classe *Activity* dispone di un metodo definito al seguente modo:

```
public boolean onCreateOptionsMenu(Menu  menu)
```

Questo metodo, nel ciclo di vita dell'attività, viene richiamato automaticamente dal sistema la prima volta che l'utente preme il tasto "menu" del suo dispositivo. L'argomento passato, un oggetto di tipo *android.view.Menu*, costituisce l'options menu inizialmente vuoto. Ridefinendo il metodo è possibile intercettare queste chiamate e popolare così il menu fornito con le voci utili alla propria applicazione. Il metodo *onCreateOptionsMenu()*, al termine dei propri compiti, deve restituire un booleano: *true* per rendere attivo il menu realizzato, *false* per dichiarare che l'attività non dispone di un menu, e quindi alla pressione del tasto "menu" del dispositivo non si deve mostrare nulla. Programmando nel corpo del metodo, il proprio options menu può essere assemblato servendosi dei metodi messi a disposizione dagli oggetti di tipo *android.view.Menu*. Aggiungere un elemento al menu è molto semplice, basta servirsi dei metodi:

```
public MenuItem add(CharSequence title)
public MenuItem add(int titleRes)
```

Entrambi i metodi aggiungono un elemento al menu. Il titolo dell'elemento (cioè la scritta che sarà mostrata per qualificarlo) può essere espressa con una stringa, nel primo caso, o con il riferimento ad una risorsa di tipo stringa memorizzata su file XML esterno, nel secondo caso. Un controllo più granulare sugli elementi inseriti è reso possibile da queste altre due varianti del metodo *add()*:

```
public MenuItem add(int groupId, int itemId, int
                                   order, CharSequence title)
public MenuItem add(int groupId, int itemId, int
                                   order, int titleRes)
```

In questo caso, oltre al titolo, è possibile specificare altre tre proprietà di ciascun elemento del menu:

- ***groupId***
  Con questo valore è possibile assegnare l'elemento ad un gruppo. Si può specificare un qualsiasi valore maggiore di zero (ovviamente deve essere lo stesso per due o più elementi che si intende assegnare al medesimo gruppo), oppure si può usare lo zero o la costante *Menu.NONE* se non si vuole assegnare l'elemento ad alcun gruppo.

- ***itemId***
  Con questo valore si assegna un identificativo all'elemento stesso. Questo identificativo torna utile in seguito, quando si vuole distinguere un elemento da un altro. Come nel caso precedente, bisogna usare un valore maggiore di zero affinché la caratteristica possa essere sfruttata. Se non si è interessati all'assegnare un identificativo all'elemento, è sufficiente usare il valore zero o la costante *Menu.NONE*.

- ***order***
  Se si vuole assegnare uno specifico ordine all'elemento, è possibile in questo caso specificarlo esplicitamente con un valore da uno in su. Usando lo zero o la costante *Menu.NONE* si lascia stabilire al sistema l'ordine dell'elemento nel menu di appartenenza.

Sperimentiamo subito quanto descritto finora. Realizziamo un'attività dimostrativa così formulata:

```
package it.ioprogrammo.menudemo01;
import android.app.Activity;
import android.view.Menu;
public class MenuDemo01Activity extends Activity {
...
```

Abbiamo popolato il menu con sei semplici comandi, etichettati rispettivamente "*Comando 1*", "*Comando 2*", "*Comando 3*" e così via. Installate l'applicazione su un dispositivo e provate il suo menu. Dovreste ottenere un risultato simile a quello in fig. 5. Come potete osservare, i sei elementi introdotti sono andati a costituire l'icon menu dell'attività. Provate ora la seguente variante:

Fig. 2: L'expanded menu del browser di Android

Fig. 3: Il context menu del browser di Android visualizzato quando si tiene il tocco per qualche secondo sopra ad un'immagine contenuta in una pagina Web

Fig. 4: Un submenu del browser di Android

NOTA

ANDROID 2.1 (API 7)

Anche questo mese c'è da segnalare una nuova release di Android, la 2.1. Le API vengono avanzate al livello 7. Si tratta di una release minore, e le modifiche per quel che riguarda la programmazione sono poche. Se volete dare un'occhiata al changelog: http://developer.android.com/sdk/android-2.1.html

Fig. 5: Un semplice icon menu, con sei comandi testuali

Fig. 9: Sono stati aggiunti nove comandi al menu. I primi cinque sono entrati a far parte dell'icon menu, mentre i restanti quattro sono stati posizionati nell'expanded menu dell'attività

```
package it.ioprogrammo.menudemo02;
import android.app.Activity;
import android.view.Menu;
public class MenuDemo02Activity extends Activity {
...
```

In questo caso i comandi sono nove. Lanciate l'attività e verificate cosa accade. Mentre nel caso precedente tutti e sei i comandi previsti sono diventati parte dell'icon menu dell'attività, ora le cose sono andate diversamente. Come accennato in apertura, un icon menu può contenere al massimo sei elementi, come nel caso del primo esempio. Ora che gli elementi sono diventati nove, il sistema ha posizionato nell'icon menu solo i primi cinque del lotto. Il sesto spazio disponibile nell'icon menu è stato automaticamente occupato con un elemento di tipo "altro", che lavora come pulsante d'accesso per l'expanded menu dell'attività. In quest'ultimo sono stati inseriti i restanti quattro comandi. Gli elementi che confluiscono nell'icon menu possono utilizzare delle icone al posto del testo. Usufruire di questa funzionalità è molto semplice. Per prima cosa dovete fare caso al fatto che i metodi *add()* di *Menu* restituiscono un oggetto di tipo *android.view.MenuItem*. Come è facile intuire, l'oggetto restituito rappresenta l'elemento appena introdotto nel menu. Questo genere di oggetti dispone di metodi che permettono il controllo dell'elemento. Fra questi segnaliamo i seguenti:

```
public MenuItem setIcon(Drawable icon)
public MenuItem setIcon(int iconRes)
```

Ambo i metodi servono per aggiungere un'icona all'elemento. Si può usare un oggetto *graphics.drawable.Drawable*, caricato o realizzato in precedenza, oppure il riferimento ad un'immagine conservata nella directory *res/drawable* del progetto. Una volta impostata un'icona su un elemento, questa sarà mostrata solo se l'elemento è parte dell'icon menu. Ecco un esempio che dimostra la funzionalità:

```
package it.ioprogrammo.menudemo03;
...
public class MenuDemo03Activity extends Activity {
...
```

Affinché l'esempio funzioni correttamente, è necessario disporre delle immagini *play*, *pause* e *stop* nella directory *res/drawable* del progetto. Nel CD-Rom allegato alla rivista troverete tutto ciò di cui avrete bisogno. Una volta fatta funzionare l'attività, il suo icon menu sarà come quello mostrato in Fig. 7. Ora che abbiamo imparato a disporre gli elementi nell'options menu, impariamo anche come gestire gli eventi di attivazione di ciascuno degli elementi introdotti. Esistono un paio di maniere per intercettare gli eventi di tocco riscontrati sugli elementi di un options menu. La prima tecnica consiste nel ridefinire un metodo di *Activity* così definito:

```
public boolean onOptionsItemSelected(MenuItem item)
```

Questo metodo viene richiamato automaticamente ogni volta che uno degli elementi dell'options menu dell'attività viene selezionato dall'utente. Lo specifico elemento selezionato, naturalmente, è quello riportato in argomento, mentre il valore di ritorno serve per indicare al sistema se l'evento è stato gestito (*true*) oppure no (*false*). Ridefinendo il metodo e applicando dei filtri sull'identificativo dell'elemento segnalato (è possibile recuperarlo con il metodo *getId()* di *MenuItem*) è possibile riconoscere e gestire la specifica azione eseguita dall'utente. Ecco un esempio che mostra un avviso differente in base alla voce di menu selezionata:

```
package it.ioprogrammo.menudemo04;
...
public class MenuDemo04Activity extends Activity {
...
```

Questo stralcio di codice mette in atto una tecnica consigliata: memorizzare gli identificativi degli elementi del menu con delle costanti. In questo modo gli *ID* risultano più leggibili e si è meno inclini a commettere errori di battitura o distrazione. Una seconda tecnica per la gestione degli eventi consiste nell'utilizzare il seguente metodo di *MenuItem*:

```
public MenuItem setOnMenuItemClickListener
        (MenuItem.OnMenuItemClickListener
                menuItemClickListener)
```

La tecnica richiama le logiche di gestione degli eventi che abbiamo conosciuto il mese scorso. L'interfaccia *MenuItem.OnMenuItemClickListener* richiede l'implementazione del metodo:

```
public boolean onMenuItemClick(MenuItem item)
```

Il metodo viene richiamato quando l'elemento è selezionato dall'utente. Ecco un esempio analogo al precedente, ma basato su questa seconda tecnica di gestione degli eventi:

```
package it.ioprogrammo.menudemo05;
...
public class MenuDemo05Activity extends Activity {
...
```

La classe *Activity*, infine, dispone di altri due metodi collegati alla gestione del suo options menu:

- ***public boolean onPrepareOptionsMenu(Menu menu)***

Viene richiamato quando l'options menu sta per essere visualizzato. Infatti il metodo *onCreateOptionsMenu()*, usato in tutti gli esempi precedenti, viene richiamato solo la prima volta che il menu deve essere mostrato. Se si intende modificare il menu costruito in precedenza, si può ridefinire questo secondo metodo. Gli oggetti *Menu*, a tal proposito, dispongono di una serie di metodi che permettono di recuperare, modificare e rimuovere gli elementi introdotti al suo interno in precedenza. Anche se non è un caso molto comune, quindi, Android permette lo sviluppo di options menu dinamici, che cambiano in base allo stato dell'applicazione.

- ***public void onOptionsMenuClosed(Menu menu)***
  Viene richiamato quando l'options menu, dopo essere stato visualizzato, viene chiuso.

- ***public void openOptionsMenu()***
  Apre il menu automaticamente, senza che sia necessario premere il tasto "*menu*" del dispositivo.

CONTEXT MENU

I menu contestuali permettono di associare particolari opzioni o comandi ai singoli widget di un'attività. La creazione e l'utilizzo dei context menu sono molto simili a quelli dell'options menu.
La prima cosa che si deve fare è dichiarare che uno o più widget dell'attività devono disporre di un menu contestuale. Lo si può fare con il metodo *registerForContextMenu(View view)* di *Activity*.
Ad esempio, per dotare un bottone di un menu contestuale si deve fare così:

```
Button mioBottone = new Button(this);
// ...
registerForContextMenu(mioBottone);
```

Se il widget è stato dichiarato in un XML di layout, lo si può caricare attraverso il suo identificativo:

```
View mioWidget = findViewById(R.id.mioWidgetId);
// ...
registerForContextMenu(mioBottone);
```

Fatto ciò, è possibile ridefinire uno o più metodi di *Activity* in modo da generare e gestire il menu contestuale. Questi metodi sono:

- ***public void onCreateContextMenu(ContextMenu menu, View v, ContextMenu.ContextMenuInfo menuInfo)***
  Questo metodo viene richiamato ogni volta che il menu contestuale per il widget *v* deve essere mostrato. Il menu deve essere costruito aggiungendo elementi al parametro *menu*. Il terzo argomento, *menuInfo*, viene usato solo in alcuni casi che esamineremo in futuro, come ad esempio quando si ha a che fare con le liste.

- ***public boolean onContextItemSelected(MenuItem item)***
  Richiamato quando un elemento di un context menu viene selezionato.
- ***public void onContextMenuClosed(Menu menu)***
  Richiamato quando il menu contestuale viene chiuso.

Gli oggetti *android.view.ContextMenu* dispongono di tutti i metodi già visti con gli oggetti *Menu*. Fate però attenzione al fatto che gli elementi dei menu contestuali non supportano né le icone né le scorciatoie da tastiera (cfr. box laterale).
In compenso gli oggetti *ContextMenu* si specializzano attraverso i seguenti metodi:

- ***public ContextMenu setHeaderTitle(CharSequence title)***
  public ContextMenu setHeaderTitle(int titleRes)
  Associa un titolo al menu contestuale, che sarà mostrato nell'intestazione del menu.

- ***public ContextMenu setHeaderIcon(Drawable icon)***
  public ContextMenu setHeaderIcon(int iconRes)
  Associa un'icona al menu contestuale, che sarà mostrata nell'intestazione del menu.

- ***public ContextMenu setHeaderView(View view)***
  Imposta l'oggetto *view* come intestazione del menu, sostituendo l'icona ed il titolo dei metodi precedenti.

- ***public void clearHeader()***
  Ripulisce l'intestazione del menu.

Realizziamo insieme il seguente esempio di codice:

```
package it.ioprogrammo.menudemo06;
...
public class MenuDemo06Activity extends Activity {
...
```

Si tratta di un'attività che dispone di due bottoni, tutti e due collegati ad un menu contestuale. L'esempio mostra come registrare i widget per il menu contestuale, come creare menu distinti per widget differenti e come gestire gli eventi dei menu contestuali. A proposito di eventi: anche nel caso dei context menu è possibile sfruttare la tecnica di gestione alternativa basata sugli oggetti *OnMenuItemClickListener*.

LA REGOLA DEI TRE CLIC

Una vecchia regola di usabilità dice che l'utente deve sempre poter trovare quel che cerca con al massimo tre clic. Se non ci riesce, l'utente inizia a provare frustrazione. Questa regola è stata inizialmente concepita per il Web ma, di fatto, oggi può essere applicata anche alle applicazioni mobili e desktop di tipo non professionale e dedicate ad un pubblico quanto più ampio possibile. I menu di Android, è evidente, sono stati concepiti tenendo a mente la regola.
Per approfondire: http://en.wikipedia.org/wiki/Three-click_rule

Fig. 7: Le icone degli elementi vengono mostrate nell'icon menu.

SUBMENU

Aggiungere sotto-menu ad un options menu o ad un context menu è possibile grazie ai seguenti metodi, disponibili per oggetti *Menu* e *ContextMenu*:

- ***public SubMenu addSubMenu(CharSequence title)***
- ***public SubMenu addSubMenu(int titleRes)***
- ***public SubMenu addSubMenu(int groupId, int itemId, int order, CharSequence title)***
- ***public SubMenu addSubMenu(int groupId, int itemId, int order, int titleRes)***

Questi metodi assomigliano molto ai metodi *add()* per l'aggiunta di un comune elemento, ed infatti funzionano alla stessa maniera. La differenza è che l'elemento inserito, una volta selezionato, causerà l'apertura di un sotto-menu. Cosa sarà mostrato nel sotto-menu possiamo stabilirlo con l'oggetto *android.view.SubMenu*, restituito dai metodi *addSubMenu()*. Anche in questo caso avremo a disposizione tutti i metodi di *Menu*, ed avremo a disposizione anche dei metodi come quelli di *ContextMenu* per la definizione di un'intestazione del menu. In effetti i submenu somigliano molto ai context menu:

```
package it.ioprogrammo.menudemo07;
...
public class MenuDemo07Activity extends Activity {
...
```

NOTA

ABILITARE E DISABILITARE GLI ELEMENTI

Gli elementi di un menu possono essere disabilitati e successivamente riabilitati servendosi del seguente metodo di *MenuItem*:
public MenuItem setEnabled(boolean enabled)
Quando un elemento è disabilitato, l'utente non può selezionarlo: come se non ci fosse.

L'AUTORE

Carlo Pelliccia lavora presso 4IT (www.4it.it), dove si occupa di analisi e sviluppo software per piattaforme Java. Nella sua carriera di technical writer ha pubblicato cinque manuali ed oltre centocinquanta articoli, molti dei quali proprio tra le pagine di ioProgrammo. Il suo sito, che ospita anche diversi progetti Java Open Source, è disponibile all'indirizzo www.sauronsoftware.it

GRUPPI DI ELEMENTI

Gli elementi di un menu possono essere riuniti in dei gruppi. Due o più elementi sono nello stesso gruppo se, quando li si è aggiunti, si è usato lo stesso valore groupId. Ad esempio:

```
menu.add(1, MENUITEM_COMANDO_1, 1, "Comando 1");
menu.add(1, MENUITEM_COMANDO_2, 1, "Comando 2");
menu.add(Menu.NONE, MENUITEM_COMANDO_3, 3,
                                        "Comando 3");
```

"*Comando 1*" e "*Comando 2*" appartengono al gruppo con *groupId 1*, mentre "*Comando 3*" non fa parte di alcun gruppo (il suo *groupId* è *Menu.NONE*). I gruppi di elementi permettono di velocizzare alcune operazioni. Ad esempio se si intende abilitare o disabilitare tutti gli elementi di un gruppo, è sufficiente richiamare il metodo di *Menu* così definito:

```
public void setGroupEnabled(int groupId, boolean
                                          enabled)
```

Gli elementi di un gruppo possono essere resi visibili o invisibili chiamando il metodo:

```
public void setGroupVisible(int groupId, boolean visible)
```

Più particolare il seguente metodo:

```
public void setGroupCheckable(int group, boolean
                       checkable, boolean exclusive)
```

Questo metodo rende *checkable* tutti gli elementi di un gruppo. Quando un elemento è *checkable*, si comporta come un interruttore che può essere acceso (selezionato) o spento (non selezionato). Se il parametro *exclusive* è *false*, gli elementi si comporteranno come delle checkbox: quindi sarà possibile selezionarne anche due o più contemporaneamente. Se *exclusive* è *true*, invece, gli elementi del gruppo si comporteranno come dei bottoni radio. Si può selezionare o deselezionare un elemento con il metodo di *MenuItem* così definito:

```
public MenuItem setChecked(boolean checked)
```

Se un elemento è selezionato oppure no, invece, lo si può sapere chiamando:

```
public boolean isChecked()
```

Ecco un esempio che dimostra questa funzionalità:

```
package it.ioprogrammo.menudemo08;
...
public class MenuDemo08Activity extends Activity {
...
```

MENU IN SALSA XML

I menu, proprio come i layout, possono essere definiti via XML. Per farlo si usa la speciale cartella *res/menu*. I file XML al suo interno usano i tag *<menu>*, *<group>* e *<item>* per definire i menu in modo dichiarativo. Eclipse, attraverso il plug-in per lo sviluppo Android, dispone di un plug-in per l'editing visuale di questi file. I menu XML possono essere caricati attraverso la classe *android.view.MenuInflater* ed il suo metodo *inflate(int menuRes, Menu menu)*. Terminiamo con il caso di caricare il menu */res/menu/menu1.xml* come options menu di un'attività:

```
@Override
public boolean onCreateOptionsMenu(Menu menu)
{
   MenuInflater inflater = new MenuInflater(this);
   inflater.inflate(R.menu.menu1, menu);
   return true;
}
```

Carlo Pelliccia

NOTIFICHE E FINESTRE DI DIALOGO

OTTAVO APPUNTAMENTO. QUESTO MESE INCREMENTEREMO L'INTERATTIVITÀ DELLE NOSTRE APPLICAZIONI, DOTANDOLE DELLA POSSIBILITÀ DI EMETTERE DEGLI AVVISI E DI INTERROGARE L'UTENTE ATTRAVERSO LE FINESTRE DI DIALOGO

Nei mesi passati abbiamo già appreso numerose tecniche per dialogare con chi utilizza l'applicazione: tra widget, eventi e menu siamo già in grado di costruire applicazioni interattive. Ci sono comunque altri due strumenti che non possono assolutamente mancare nel nostro armamentario: sono i cosiddetti *toast* e le finestre di dialogo. I primi servono per segnalare delle notifiche, mentre le seconde possono essere usate sia per emettere un output sia per ricevere un input.

UN TOAST COME AVVISO

Un toast è un avviso mostrato per qualche istante in sovraimpressione sullo schermo. Le notifiche toast vanno usate per brevi messaggi testuali. Insomma, informazioni del tipo "impostazioni salvate", "operazione eseguita" e simili. I messaggi toast rimangono sullo schermo per qualche istante e poi il sistema li rimuove automaticamente: non c'è alcuna interazione con l'utente. La classe di riferimento per la gestione dei messaggi toast è *android.widget.Toast*. A disposizione ci sono i seguenti due metodi statici:

- ***public static Toast makeText(Context context, CharSequence text, int duration)***
- ***public static Toast makeText(Context context, int resId, int duration)***

Entrambi i metodi costruiscono un messaggio toast testuale. I parametri da fornire sono, rispettivamente, il contesto applicativo (ad esempio l'attività stessa), il messaggio da mostrare (come stringa, nel primo caso, o come riferimento a risorsa esterna, nel secondo) e la durata del messaggio. Non è possibile specificare quanti secondi, esattamente, il messaggio dovrà restare visibile. Si può soltanto dire se il messaggio deve durare poco o tanto. Per farlo si deve usare come argomento duration una fra le due costanti *Toast.LENGTH_SHORT* (durata breve) o *Toast.LENGTH_ LONG* (durata lunga). Ecco un esempio:

Fig. 1: Ecco come appare un messaggio toast nell'interfaccia di Android

```
Toast toast = Toast.makeText(this, "Questo è un toast",
                                   Toast.LENGTH_LONG);
```

Una volta creato, il toast può essere mostrato chiamandone il metodo *show()*:

```
toast.show();
```

Altri metodi degli oggetti *Toast* permettono di impostare ulteriori proprietà dell'avviso. Si consideri ad esempio il metodo:

public void setGravity(int gravity, int xOffset, int yOffset)

Con questo metodo si può impostare in quale angolo dello schermo far apparire il messaggio (*gravity*, cfr. ioProgrammo 146), specificando anche il discostamento dai bordi laterali (*xOffset*) e da quelli verticali (*yOffset*). Ad esempio:

```
toast.setGravity(Gravity.TOP | Gravity.LEFT, 10, 10);
```

Questo avviso sarà mostrato in alto a sinistra, scostato di 10 pixel dai bordi. Si possono anche creare dei messaggi toast che, invece di mostrare del semplice testo, facciano uso di immagini o di altri widget al loro interno. In tal caso, invece di passare per i metodi statici *makeToast()*, si usa il costruttore della classe, che vuole in argomento il contesto dell'applicazione:

```
Toast toast = new Toast(this);
```

La durata, in questo caso, la si può stabilire con *setDuration()*:

```
toast.setDuration(Toast.LENGTH_LONG);
```

Il contenuto del toast, adesso, può essere del testo, come nel caso precedente:

```
toast.setText("messaggio di testo");
```

Esiste anche una seconda variante di *setText()* che permette l'utilizzo delle stringhe esterne:

```
toast.setText(R.string.messaggio_esterno);
```

Per un toast graficamente più ricco, invece, si può usare il metodo *setView(View view)*, che imposta il widget da visualizzare all'interno della notifica in sostituzione del testo. Ad esempio un'icona:

```
ImageView image = new ImageView(this);
image.setImageResource(R.drawable.mia_icona);

Toast toast = new Toast(this);
toast.setGravity(Gravity.CENTER, 0, 0);
toast.setDuration(Toast.LENGTH_LONG);
toast.setView(image);
toast.show();
```

Un layout XML esterno può essere caricato, proprio sotto forma di oggetto View, passando attraverso un oggetto *android.view.LayoutInflater*. Ogni attività ne mette a disposizione un'istanza:

```
LayoutInflater inflater = getLayoutInflater();
View view = inflater.inflate(R.layout.toast_xml, null);
```

Questo significa che toast di maggiore complessità possono essere creati con la più agevole sintassi di XML, per essere poi caricati dinamicamente quando occorre mostrarli.
Nel CD-Rom allegato alla rivista troverete degli esempi completi di utilizzo delle notifiche toast.

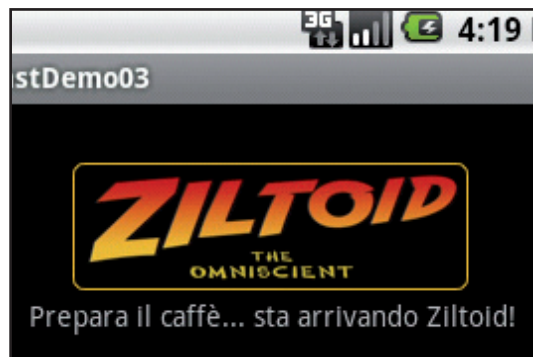



Fig. 2: Un toast di maggiore complessità, con icona e testo, costruito a partire da una definizione di layout esterna, in formato XML

FINESTRE DI DIALOGO

Le finestre di dialogo sono dei riquadri che è possibile aprire sopra l'attività corrente. Quando una finestra di dialogo compare, l'attività da cui dipende viene bloccata, e l'utente deve necessariamente interagire con la finestra di dialogo per farvi ritorno. L'esempio tipico è la finestra di conferma, che domanda all'utente se vuole proseguire con una certa operazione. L'utente, quando tale finestra compare, non può far altro che scegliere tra l'opzione per proseguire e quella per annullare. Finché la scelta non viene espressa, l'attività sottostante rimane bloccata e non può essere ripresa.
A differenza dei toast, quindi, le finestre di dialogo sono sia bloccanti sia interattive. Per questo motivo la loro gestione risulta lievemente più complessa.
L'astrazione di base cui far riferimento è la classe *android.app.Dialog*, che definisce mediante i suoi metodi cosa una finestra di dialogo può fare e come può essere manipolata. Nei prossimi paragrafi la approfondiremo più nello specifico. Adesso, invece, concentriamoci sul ciclo di vita delle finestre di dialogo, e sulla maniera che dovremo adoperare per richiamarle e mostrarle.

Fig. 3: Una richiesta di conferma all'interno di una finestra di dialogo

La classe *Activity* fornisce un metodo così definito:

public final void showDialog(int id)

Possiamo richiamare questo metodo ogni volta che dobbiamo aprire e mostrare una finestra di dialogo. Il parametro id simboleggia quale specifica finestra di dialogo l'attività deve mostrare. Il valore è arbitrario, nel senso che è nostro compito creare le finestre di dialogo ed assegnargli degli identificativi. La prassi consiglia di usare delle costanti interne alla classe dell'attività.
Mi spiego meglio attraverso un esempio. Facciamo il caso che la nostra attività faccia uso di due finestre di dialogo, una per emettere un avviso di errore ed una per richiedere una conferma. La miglior cosa da fare, in casi come questo, è aggiungere due costanti all'attività, con nomi e valori arbitrari ma univoci. Ad esempio:

SDK TOOLS R5 E ADT 0.9.6

A metà marzo è stata rilasciata la versione r5 dei tool di sviluppo Android, per Windows, Linux e MacOS X. Per sviluppare in ambiente Eclipse con la nuova versione del kit è necessario aggiornare il proprio plug-in ADT alla versione 0.9.6. Gli indirizzi di riferimento sono:

http://developer.android.com/sdk/index.html

http://developer.android.com/sdk/eclipse-adt.html

NOTA

TOAST: NON SOLO DALLE ATTIVITÀ

I messaggi toast possono essere mostrati non soltanto dalle attività, ma anche da altri tipi di applicazioni Android, come i servizi.

NOTA

INSERIRE UNA DATA

Altre due finestre di dialogo previste dalla libreria di Android sono *DatePickerDialog* e *TimePickerDialog*, entrambe nel pacchetto *android.app*. Come il loro nome lascia intuire, servono per far selezionare all'utente una data o un orario.

```
private static final int DIALOG_ERROR_ID = 1;
private static final int DIALOG_CONFIRM_ID = 2;
```

Quando dovremo mostrare l'avviso di errore, dunque, chiameremo:

```
showDialog(DIALOG_ERROR_ID);
```

Analogamente, per la richiesta di conferma, dovremo fare:

```
showDialog(DIALOG_CONFIRM_ID);
```

Adesso viene da chiedersi: come fa Android a sapere quali finestre corrispondano effettivamente ai due interi indicati? Ed infatti, allo stato attuale delle cose, Android non lo sa: siamo noi a dover svolgere le associazioni. Per farlo dobbiamo ridefinire il metodo di *Activity* avente firma:

protected Dialog onCreateDialog(int id)

Android richiama questo metodo ogni volta che deve creare una finestra di dialogo. La finestra che deve essere creata è identificata dal parametro *id*. Ridefinendo il metodo dobbiamo riconoscere l'identificativo fornito, costruire la finestra di dialogo corrispondente e restituirla sotto forma di oggetto *android.app.Dialog*. Lo schema consigliato è il seguente:

```
@Override
protected Dialog onCreateDialog(int id) {
    Dialog dialog;
    switch (id) {
    ...
```

In breve, si utilizza un costrutto *switch* per associare degli oggetti *Dialog* ai loro corrispettivi identificativi numerici. Nel codice di esempio si fa riferimento ai due metodi *createErrorDialog()* e *createConfirmDialog()*, che sono dei metodi custom che lo sviluppatore crea a proprio piacimento per generare le differenti finestre di dialogo di cui ha bisogno.
Dopo che la specifica finestra di dialogo è stata creata, Android prima di mostrarla richiama il seguente metodo di *Activity*:

protected void onPrepareDialog(int id, Dialog dialog)

I due parametri corrispondono, rispettivamente, all'identificativo della finestra e all'oggetto Dialog costruito nel passaggio precedente. Lo sviluppatore può opzionalmente ridefinire questo metodo per inizializzare la finestra di dialogo con dei comandi specifici. Il modello, questa volta, è il seguente:

```
@Override
protected void onPrepareDialog(int id, Dialog dialog) {
    switch (id) {
    ...
```

È importate sapere che il metodo *onCreateDialog()* per uno specifico id viene richiamato solo la prima volta che la corrispondente finestra di dialogo deve essere mostrata. La seconda volta che la stessa finestra dovrà essere mostrata, il sistema farà riuso dell'istanza già esistente. Se bisogna inizializzare l'istanza in maniera differente dalla volta precedente, quindi, non resta che farlo ridefinendo *onPrepare Dialog()* e adottando lo schema proposto sopra. Una volta che una finestra di dialogo viene aperta, ci sono due maniere per chiuderla. In primo luogo la può chiudere l'utente servendosi del tasto "*back*" del suo dispositivo. Non sempre è possibile farlo, dipende dal tipo e dalle impostazioni della specifica finestra di dialogo. Quando è possibile farlo si dice che la finestra è *cancelable* (cioè cancellabile). Via codice, invece, è sempre possibile chiudere e terminare qualsiasi finestra di dialogo. Lo si può fare invocando il metodo di *Activity* così definito:

public final void dismissDialog(int id)

Ad esempio:

```
dismissDialog(DIALOG_ERROR_ID);
```

La finestra dismessa viene chiusa e nascosta. Come spiegato prima, però, un riferimento all'oggetto *Dialog* che la rappresenta viene conservato all'interno dell'attività, in modo che l'istanza possa essere riusata nel caso in cui la stessa finestra debba essere nuovamente mostrata. In realtà è possibile far dimenticare del tutto la finestra di dialogo, servendosi al posto di *dismissDialog()* del metodo:

public final void removeDialog(int id)

In questo caso l'istanza della corrispondente finestra di dialogo viene eliminata. Se la finestra dovrà essere mostrata di nuovo, sarà necessario ricrearla utilizzando nuovamente *onCreateDialog()*. La prassi dunque è invocare *dismissDialog()* per le finestre di dialogo usate frequentemente, *removeDialog()* per chiudere quelle mostrate raramente. Ora che abbiamo le idee chiare sul ciclo di vita delle finestre di dialogo, impareremo come costruire e amministrare una finestra di dialogo. Poche volte si usa direttamente la classe *Dialog*: nella stragrande maggioranza dei casi si fa prima ad usare una delle sue sottoclassi messe a disposizione dalla libreria di Android. Classi come *android.app.AlertDialog* e *android.app.ProgressDialog*, infatti, coprono il 99% delle esigenze. Andiamo a conoscerle.

ALERTDIALOG

Il primo tipo di finestra di dialogo che studieremo è *android.app.AlertDialog*. È quella utile per mostrare un avviso o per chiedere qualcosa all'utente, come una conferma o la selezione di un elemento da una lista. Cominciamo dal più semplice dei casi: vogliamo notificare un evento e vogliamo essere sicuri che l'utente ne prenda atto. Un messaggio toast, in questo caso, non andrebbe bene: potrebbe scomparire prima che l'utente lo noti. Useremo allora una finestra di dialogo in grado di bloccare l'applicazione fin quando l'utente non noterà ed accetterà il messaggio. Il codice per farlo è molto semplice:

```
AlertDialog.Builder builder = new
                    AlertDialog.Builder(this);
builder.setTitle("Avviso");
builder.setMessage("Attenzione! Questo è un avviso!");
builder.setCancelable(true);
AlertDialog alert = builder.create();
```

Come è possibile osservare, la finestra di dialogo viene prodotta servendosi di uno speciale oggetto *AlertDialog.Builder*. A questo oggetto builder si deve dire quali sono le caratteristiche dell'*AlertDialog* desiderato, e per farlo sono a disposizione numerosi metodi. In questo caso abbiamo specificato il titolo, con *setTitle()*, ed il messaggio, con *setMessage()*. Con il comando *setCancelable(true)* abbiamo fatto in modo che l'avviso possa essere chiuso con il tasto "back" del telefono. Il metodo *create()*, a questo punto, è stato invocato per produrre la finestra di dialogo. Una finestra di dialogo così realizzata, adesso, potrebbe essere restituita dal metodo *onCreateDialog()*, producendo un risultato come quello in Fig.4. Facciamo ora il caso di voler produrre un avviso identico al precedente, ma che, invece di costringere l'utente ad usare il tasto "*back*" del dispositivo, metta a disposizione esso stesso un bottone "*chiudi*". Ecco come fare:

```
AlertDialog.Builder builder = new
                    AlertDialog.Builder(this);
builder.setTitle("Avviso");
builder.setMessage("Attenzione! Questo è un avviso!");
builder.setCancelable(false);
builder.setPositiveButton("Chiudi", new
...
```

Con *setCancelable(false)* abbiamo disabilitato il tasto fisico del dispositivo, mentre con *setPositiveButton()* abbiamo aggiunto il bottone "*chiudi*". Al metodo abbiamo dovuto anche fornire un oggetto di tipo *android.content.DialogInterface. OnClickListener*. Si tratta, come è facile intuire, di un gestore di eventi richiamato alla pressione del tasto "*chiudi*". Con una chiamata a *dismissDialog()* facciamo dunque in modo che la finestra di dialogo venga chiusa quando l'utente tocca il bottone. Semplice, vero? Altrettanto semplice, a questo punto, è creare una richiesta di conferma, come quella discussa in apertura di paragrafo:

```
AlertDialog.Builder builder = new
                    AlertDialog.Builder(this);
builder.setTitle("Conferma");
builder.setMessage("Vuoi proseguire con l'operazione?");
builder.setCancelable(false);
builder.setPositiveButton("Prosegui", new
               DialogInterface.OnClickListener() {
    @Override
    ...
```

Alla risposta positiva programmata con *setPositiveButton()*, abbiamo aggiunto ora una risposta negativa, con *setNegativeButton()*. Il metodo è simile al precedente: anche in questo caso dobbiamo fornire un listener che intercetti la pressione sul bottone e gestisca di conseguenza l'evento.

E se volessimo fornire la facoltà di scegliere fra più di due opzioni? Anche questo è possibile:

```
final String[] options = { "Caffè", "Gelato", "Tè",
                    "Birra", "Ragù" };
AlertDialog.Builder builder = new
                    AlertDialog.Builder(this);
builder.setTitle("Scelta multipla");
builder.setItems(options, new
...
```

In questo caso non si sono usati né *setMessage()* né i metodi *setPositiveButton()* e *setNegativeButton()*. Si è invece fatto ricorso al metodo *setItems()*. Questo metodo vuole come argomento un array di stringhe. Ciascuna delle stringhe fornite sarà un'opzione di scelta. Ancora una volta, il secondo argomento da fornire è il gestore di eventi. Quando una delle opzioni sarà selezionata dall'utente, il metodo *onClick()* del gestore verrà automaticamente richiamato. L'argomento *which*, in questo caso, riporterà l'indice dell'opzione selezionata. L'elenco di opzioni

Fig. 4: Un avviso che può essere chiuso con il tasto "back" del telefono

NOTIFICHE NELLA STATUS BAR

La status bar (o barra di stato) è la barra posizionata nella parte alta dello schermo, quella che contiene l'orologio, per intenderci. La barra di stato di Android viene utilizzata, tra le altre cose, anche per trasmettere delle notifiche all'utente. Ad esempio quando arriva un SMS la barra si attiva e mostra un'icona che segnala l'evento all'utente. L'utente può poi "srotolare" la barra ed esaminare i dettagli della notifica ricevuta. Anche le applicazioni custom possono segnalare delle notifiche all'utente utilizzando questo sistema. Più che le attività, ad ogni modo, in genere sono i servizi a farlo. Per approfondire:

http://developer.android.com/guide/topics/ui/notifiers/notifications.html

Fig. 5: Un avviso che può essere chiuso attraverso il suo stesso bottone "chiudi"

Fig. 6: Una richiesta di conferma con due bottoni

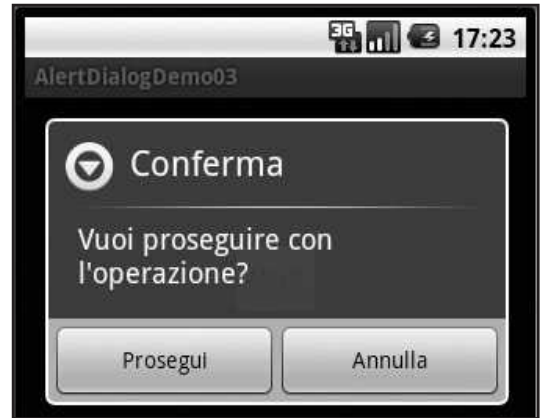
Fig. 7: Un AlertDialog con numerose opzioni

Fig. 8: Un AlertDialog con le opzioni rese come bottoni radio

Fig. 9: Un AlertDialog con le opzioni rese come checkbox, per consentire una scelta multipla

può essere anche popolato con dei bottoni radio o con delle caselle di tipo checkbox. Nel primo caso si deve utilizzare, al posto di *setItems()*, il metodo *setSingleChoiceItems()*. Questo metodo vuole tre parametri: l'array di stringhe con le opzioni, l'indice dell'opzione inizialmente selezionata e, infine, il solito gestore di eventi. Ad esempio:

```
final String[] options = { "Caffè", "Gelato", "Tè",
                    "Birra", "Ragù" };
AlertDialog.Builder builder = new
                    AlertDialog.Builder(this);
builder.setTitle("Scelta multipla");
builder.setSingleChoiceItems(options, 2, new
                    DialogInterface.OnClickListener() {
  ...
```

In questo caso si è tornati ad aggiungere i tasti di conferma e di cancellazione. Ciò ha senso: l'utente seleziona nell'elenco l'opzione desiderata e quindi conferma la sua scelta con l'apposito bottone. Visto che i listener, in questo caso, crescono di numero, si deve fare attenzione a mettere in atto una corretta politica di gestione degli eventi. La scelta multipla, infine, è possibile usando dei checkbox. Il metodo utile, in questo caso, è *setMultiChoiceItems()*. Il metodo chiede tre parametri. Il primo è la lista delle opzioni, così come la abbiamo già conosciuta nei due casi precedenti. Il secondo argomento è un array di booleani, i cui elementi corrispondono a ciascuna delle opzioni possibili, indicando se l'opzione corrispondente è inizialmente selezionata o meno. Il terzo argomento è il gestore degli eventi. Questa volta l'interfaccia è *DialogInterface.OnMultiChoiceClickListener*. Il metodo *onClick()* di questa interfaccia si differenzia da quello di *OnClickListener* perché prevede un terzo parametro. Si tratta di un booleano chiamato *isChecked*, che serve a indicare se l'opzione toccata dall'utente è stata selezionata o meno. Ecco un esempio di quanto detto:

```
final String[] options = { "Caffè", "Gelato", "Tè",
                    "Birra", "Ragù" };
final boolean[] selections = { true, false, false, false,
                    false };
AlertDialog.Builder builder = new lertDialog.Builder(this);
  ...
```

Nel CD-Rom troverete numerosi esempi completi relativi alle finestre *AlertDialog*, che riassumono e dimostrano quanto spiegato in questo paragrafo.

PROGRESSDIALOG

Può capitare che sia necessario svolgere delle operazioni non istantanee, che possono cioè durare qualche secondo o anche di più. Quando avviene ciò, si deve far capire all'utente che c'è un'operazione in corso, e che bisogna attendere fiduciosi. Se non lo si fa, l'utente potrebbe pensare che l'applicazione non gli sta rispondendo perché è lenta o, peggio ancora, perché si è bloccata.
Questo, naturalmente, è male. Per fortuna in casi come questo ci si può servire di una *android.app.ProgressDialog*. Si tratta di una finestra di dialogo concepita appositamente per mettere in attesa l'utente. Lo scopo è duplice: da una parte blocca l'attività, in modo che non si possa far altro che attendere, mentre allo stesso tempo comunica all'utente che l'applicazione sta lavorando alacremente e che tutto procede come previsto. Opzionalmente si può mostrare anche il progresso dell'operazione. Ad esempio durante un download è possibile far vedere la percentuale di completamento raggiunta. Quando la barra di avanzamento non viene mostrata si dice che la *ProgressDialog* è indeterminata. La maniera più facile per creare una *ProgressDialog* indeterminata è attraverso il suo metodo statico:

public static ProgressDialog show(Context context, CharSequence title, CharSequence message)

Il metodo crea e restituisce una finestra di attesa indeterminata, quindi senza barra di progresso. Richiede come parametri il contesto dell'applicazione (tipicamente l'attività stessa che sta creando la finestra), il titolo da assegnare alla finestra ed il messaggio da mostrare al suo interno. Le *ProgressDialog* con barra di avanzamento sono leggermente più complesse. L'oggetto, in questo caso, va costruito manualmente richiamando il costruttore:

```
ProgressDialog progress = new ProgressDialog(this);
```

Bisogna poi specificare che la barra che si sta creando non è indeterminata:

```
progress.setIndeterminate(false);
```

Adesso si deve indicare di usare la barra orizzontale:

```
progress.setProgressStyle(ProgressDialog.STYLE_HORIZ
ONTAL);
```

L'avanzamento raggiunto deve essere espresso mediante un valore che va da 0 a 100. Se, per qualche motivo, questo range non fosse adatto ad una vostra esigenza, potete cambiarne il limite superiore, invocando il metodo *setMax()*. Ad esempio:

```
progress.setMax(1000);
```

Così facendo, il valore di progresso raggiunto potrà essere espresso nel range che va da 0 a 1000.
A questo punto, la finestra di dialogo può essere restituita e visualizzata. Su un thread parallelo bisogna intanto iniziare a svolgere l'operazione di cui l'utente attende la conclusione. Di tanto in tanto, mentre si svolge tale operazione, è necessario aggiornare la barra di avanzamento della *ProgressDialog*, per informare l'utente circa il punto raggiunto. Per farlo è disponibile il metodo *setProgress()*, che accetta come parametro un valore intero che rappresenta il livello di completamento raggiunto. Ad esempio:

```
progress.setProgress(50);
```

Il valore espresso deve essere compreso nel range di default (da 0 a 100) o in quello esplicitamente modificato in precedenza chiamando *setMax()*. Per le operazioni più complesse si può addirittura usare una barra di progresso secondaria. Facciamo il caso di un'applicazione che scarica dei file da Internet. Ad un certo punto deve scaricare dieci file. In questo caso si può usare la barra di avanzamento principale per far vedere quanti file sono stati già scaricati, e la barra secondaria per mostrare il progresso raggiunto dal singolo file che di volta in volta viene scaricato. Il metodo utile è *setSecondaryProgress()*, che accetta un intero compreso tra 0 ed il valore massimo previsto. Ad esempio:

```
progress.setSecondaryProgress(30);
```

FINESTRE DI DIALOGO CUSTOM

Se *AlertDialog* e *ProgressDialog* non dovessero andare bene per una vostra specifica esigenza, potete sempre procedere alla costruzione e all'utilizzo di una finestra di dialog custom, cioè i cui contenuti sono stabiliti in tutto e per tutto da voi. Vediamo insieme come procedere. Ancora una volta, la miglior cosa da farsi è realizzare un layout XML. Realizziamo insieme il seguente:

```
<LinearLayout xmlns:android="http://schemas
.android.com/apk/res/android"
...
```

Chiamate il file *custom_dialog.xml* e disponetelo nella cartella *res/layout* del vostro progetto Android. Questo layout mette insieme un'immagine ed un testo. Né l'immagine né il testo, però, sono specificati a livello di XML: realizzeremo ora una classe estensione di *Dialog* che permetterà di impostare l'uno e l'altra. Chiamiamola *Custom Dialog*:

```
import android.app.Dialog;
import android.content.Context;
import android.os.Bundle;
import android.widget.ImageView;
import android.widget.TextView;
public class CustomDialog extends Dialog {...
```

Come potete vedere, estendere *Dialog* non è poi tanto diverso da estendere *Activity*. All'atto di creazione della finestra abbiamo provveduto affinché il layout XML realizzato in precedenza venga caricato e mostrato all'interno della finestra. Abbiamo poi predisposto i metodi *setImage()* e *setMessage()*, che impostano l'immagine ed il testo visualizzati nel layout. Ora possiamo utilizzare la classe *CustomDialog* in un'attività. Basterà fare qualcosa del tipo:

```
private static final int DIALOG_CUSTOM_ID = 1;
@Override
protected Dialog onCreateDialog(int id) {...}
@Override
protected void onPrepareDialog(int id, Dialog dialog) {
...
```

Nel CD-Rom è disponibile l'esempio completo.

Carlo Pelliccia

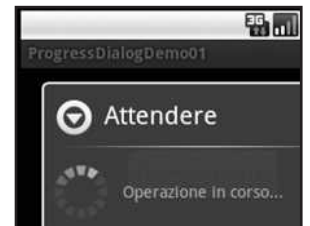



Fig. 10: Una ProgressDialog indeterminata (cioè senza barra di avanzamento)

Fig. 11: Una ProgressDialog con barra di avanzamento secondaria

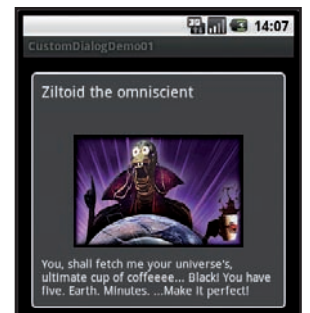



Fig. 12: Una finestra di dialogo completamente custom

Carlo Pelliccia lavora presso 4IT (www.4it.it), dove si occupa di analisi e sviluppo software per piattaforme Java. Nella sua carriera di technical writer ha pubblicato cinque manuali ed oltre centocinquanta articoli, molti dei quali proprio tra le pagine di ioProgrammo. Il suo sito, che ospita anche diversi progetti Java Open Source, è disponibile all'indirizzo www.sauronsoftware.it

INFO E FOTO: COSÌ LE PRESENTI MEGLIO!

NONO APPUNTAMENTO. CI OCCUPEREMO DEI WIDGET IN GRADO DI LEGGERE LE INFORMAZIONI DA ORGANIZZARE E MOSTRARE ALL'UTENTE. SCOPRIREMO I COMPONENTI UTILIZZATI PER REALIZZARE LISTE, TABELLE E GALLERIE DI IMMAGINI

Tutti i layout dimostrati negli esempi dei mesi precedenti sono dei layout "fissi". I widget di un layout fisso sono sempre gli stessi e non cambiano ruolo nel corso del tempo: ogni volta che si avvia l'attività, i componenti mostrati sono sempre gli stessi e la schermata, di conseguenza, appare sempre uguale. Ripensate, ad esempio, a quando abbiamo dimostrato l'uso di etichette, bottoni e caselle di testo realizzando un form per l'immissione delle generalità anagrafiche (nome, cognome, sesso) dell'utente. È stato fatto nei numeri 146 e 147 della rivista. Pensate ad un'applicazione come la rubrica del telefono, oppure la galleria delle immagini. Si tratta di attività il cui contenuto cambia di continuo. Se aggiungo un contatto o se scatto una nuova foto, ad esempio, ecco che nelle corrispondenti applicazioni apparirà un nuovo elemento.

In questo genere di applicazioni, quindi, il numero ed il tipo dei widget presenti non è sempre lo stesso, ma dipende da una fonte di informazioni esterna. La galleria delle immagini, ad esempio, costruisce e mostra tanti widget quanti sono quelli necessari per mostrare i file immagine presenti in memoria. Con le nozioni acquisite finora siamo in grado di costruire un adattatore di questo tipo. Sarebbe sufficiente, all'interno di un'attività, scrivere un ciclo *for* o qualcosa di analogo che, per ogni elemento riscontrato nella fonte dei dati, vada ad aggiungere al contenitore corrente tutti i widget necessari per mostrarlo. Qualcosa come:

```
@Override
public void onCreate(Bundle savedInstanceState) { ... }
```

Una tecnica come questa funziona (bisognerebbe usare qualche altra accortezza in più, in verità, ma l'esempio serve giusto per rendere il concetto), ma nell'ottica della programmazione Android non è il massimo. I layout XML, ad esempio, in un caso come questo non risultano facilmente applicabili. Se la schermata va costruita iterativamente, infatti, c'è poca speranza di farlo senza il supporto della logica Java. La pratica, inoltre, tende a mischiare i

***Fig. 1:** La galleria delle immagini di Android è il classico esempio di applicazione il cui layout non è fisso, in quanto il numero dei widget presenti dipende da una fonte esterna*

dati (nel caso specifico, le immagini lette dalla memoria) con lo strato dell'interfaccia utente (contenitori e widget). Ciò, secondo tutti i più moderni paradigmi della programmazione orientata agli oggetti, è male. Non vorrete mica cedere al lato oscuro della forza, vero? Per nostra fortuna la libreria di Android mette a disposizione una serie di strumenti che permettono di ottenere layout dinamici, in maniera assolutamente semplice, corretta e pulita. Basta solo imparare a utilizzarli.

ADAPTERVIEW E ADAPTER

Il primo componente che andiamo a svelare è la classe astratta *android.widget.AdapterView*. La classe estende *ViewGroup* e, pertanto, è un contenitore di altri widget. I suoi widget, però, non devono essere aggiunti facendo uso esplicito dei metodi *addView()* messi a disposizione dal contenitore. Le istanze di *AdapterView*, infatti, sono in grado di determinare da sole i loro contenuti, partendo da una sorgente esterna di informazioni che gli suggerisce quanti e quali sono gli elementi da mostrare.

Questa sorgente deve una implementazione dell'interfaccia *android.widget.Adapter*. Sono le istanze di Adapter a determinare gli elementi che devono essere mostrati, facendo da tramite tra l'attività e la fonte esterna. *AdapterView* risponderà ai comandi di Adapter, mostrando esattamente quanto richiesto. Per realizzare delle schermate di questo tipo potete estendere da voi *AdapterView* e implementare come meglio credete l'interfaccia *Adapter*, ma in verità molto raramente vi accadrà di farlo. Android dispone già di ottime implementazioni pronte all'uso, in grado di coprire la stragrande maggioranza delle esigenze di programmazione. Nei prossimi paragrafi conosceremo le implementazioni di *Adapter*, per poi applicarle insieme alle estensioni di *AdapterView* disponibili nella libreria di base.

ARRAYADAPTER

Un adattatore semplice e veloce da utilizzare è costituito dalla classe *android.widget.ArrayAdapter*. Come il nome suggerisce, si tratta di un adattatore che si comporta un po' come un array, in quanto dispone di metodi utili per aggiungere, verificare e rimuovere gli elementi al suo interno. La classe fa uso dei generics di Java per gestire meglio il tipo degli elementi al suo interno, proprio come fanno anche *java.util.ArrayList* o *java.util.Vector*. Quando si crea un oggetto di tipo *ArrayAdapter*, quindi, bisogna specificare il tipo degli elementi che conterrà. Ad esempio:

```
ArrayAdapter<String> arrayAdapter = new
        ArrayAdapter<String>(...);
```

In questo esempio gli elementi trattati dall'adattatore sono delle stringhe. Al posto delle stringhe, se necessario, potete utilizzare qualsiasi classe vi faccia comodo per il caso specifico. Fate però attenzione al fatto che gli *ArrayAdapter* sono stati concepiti per lavorare con il testo. Il loro scopo è iniettare dei widget *TextView* all'interno dell'*AdapterView* che li utilizza. I *TextView*, vi ricordo, sono dei widget testuali, utili per mostrare una stringa a schermo. Nel caso di un *ArrayAdapter* di stringhe, quindi, gli elementi aggiunti nell'adattatore saranno mostrati direttamente con delle etichette di testo. Se gli elementi gestiti, invece, non sono delle stringhe ma degli oggetti qualsiasi, nei *TextView* generati sarà introdotta la rappresentazione testuale di ciascun elemento, ottenuta richiamando il metodo *toString()* dell'oggetto corrispondente. Oltre alla classe degli elementi da gestire, *ArrayAdapter* vuole sapere anche come rappresentarli. Come accennato, *ArrayAdapter* userà una *TextView* per ciascun elemento previsto. Come questa *TextView* è fatta e in che contesto è inserita spetta allo sviluppatore deciderlo, e può farlo in XML. Si può usare un layout XML per dichiarare da quali widget sarà composto ciascun elemento della lista. L'unico vincolo è che questo layout deve necessariamente contenere un widget *TextView*. Soddisfatto questo requisito, è sufficiente notificare all'*ArrayAdapter* quale è il layout XML e quale, al suo interno, è l'id del *TextView* da utilizzare. Lo si può fare nel costruttore dell'adapter. Partendo da un layout *listitem.xml* così definito:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.
        android.com/apk/res/android"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content">
    <TextView android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:padding="5pt"
        android:textSize="10pt"
        android:id="@+id/listItemTextView" />
</LinearLayout>
```

Si può fare:

```
ArrayAdapter<String> arrayAdapter = new
        ArrayAdapter<String>(this, R.layout.listitem,
        R.id.listItemTextView);
```

Ora non resta che aggiungere gli elementi all'adapter. Il metodo suo *add()* è quel che ci vuole:

```
arrayAdapter.add("Carlo");
arrayAdapter.add("Claudia");
arrayAdapter.add("Silvia");
arrayAdapter.add("Alessandro");
```

Ovviamente inserire gli elementi della lista in questa maniera non è il massimo. Gli adattatori, infatti, servono per assorbire elementi dall'esterno, ad esempio dal file system o dal database del sistema. Di questo aspetto, naturalmente, ci occuperemo nei mesi a venire. Per ora continuiamo a concentrarci sulla classe *ArrayAdapter*. Gli elementi al suo interno possono essere recuperati con il metodo *getItem(int position)*, rimossi uno ad uno con *remove(T item)*, ripuliti completamente con *clear()* o conteggiati con *getCount()*. Da questo punto di vista, gestire un *ArrayAdapter* è proprio come gestire un comune *java.util.ArrayList*, fatta salva qualche differenza nel nome dei metodi.

IMPLEMENTARE IL PROPRIO ADAPTER

Come abbiamo appena appreso, un *ArrayAdapter* è proprio quello che ci vuole quando i dati gestiti sono una lista testuale o, comunque, rappresenta-

NOTA

GENERICS

I *generics* sono una caratteristica di Java introdotta a partire dalla versione 5 della piattaforma. L'utilizzo più comune che se ne fa è per qualificare il tipo di dato contenuto in un insieme o in una lista, evitando così la necessità di casting continuo. Prima di Java 5, infatti, un *ArrayList* di stringhe si sarebbe dovuto gestire alla seguente maniera:

```
ArrayList list = new
        ArrayList();
list.add("stringa");
String str = (String)
        list.get(i);
```

Con i generics, invece, è possibile fare:

```
ArrayList<String> list =
  newArrayList<String>();
list.add("stringa");
String str = list.get(i);
```

Per approfondire:

http://tinyurl.com/jgenerics

Fig. 2: Un esempio di ListView con ArrayAdapter

CURSORADAPTER

Un altro *Adapter* di Android molto utilizzato è *android.widget.CursorAdapter*. Non è stato illustrato nell'articolo in quanto per la sua comprensione è necessario conoscere cosa sia e come si utilizza un *Cursor*. Un cursore è l'oggetto che permette di scorrere i risultati di una query svolta al sistema di database interno ad Android, e dunque il *CursorAdapter* è un adattatore ideale per impaginare e mostrare all'utente i dati che provengono da un database. Ne riparleremo in futuro.

bile sotto forma di testo. In alcuni casi, però, questo non è vero. È il caso, per tornare su un esempio già citato, della galleria delle immagini. In questa situazione, infatti, l'adapter non deve occuparsi di testo e non deve generare widget di tipo *TextView*. Nel caso della galleria delle immagini gli oggetti da produrre saranno piuttosto dei componenti *ImageView*. Bisogna allora realizzarsi un adapter idoneo. La maniera più veloce di creare un adattatore custom consiste nell'estendere la classe astratta *android.widget.BaseAdapter*. I metodi da implementare sono:

- **public int getCount()** Questo metodo restituisce il numero di elementi presenti nell'adattatore.
- **public Object getItem(int position)** Resttituisce l'elemento alla posizione indicata.
- **public long getItemId(int position)** Restituisce un id per l'elemento alla posizione indicata.
- **public View getView(int position, View convertView, ViewGroup parent)** Questo è il metodo più importante del lotto. Deve restituire il widget che rappresenterà l'elemento sullo schermo. L'indice dell'elemento da rappresentare è dato dal parametro *position*. Il parametro *convertView* costituisce il widget generato dall'adapter ad una sua precedente chiamata. Se possibile, infatti, l'adattatore deve cercare di riciclare i widget, per risparmiare memoria. Il parametro *parent*, infine, è il contenitore che dovrà ospitare il widget generato o riciclato.

Ecco un esempio di implementazione che permette di gestire una lista di immagini (fornite sotto forma di oggetti *android.graphics.drawable.Drawable*):

```
import android.content.Context;
...
public class ImageAdapter extends BaseAdapter {... }
```

LA CLASSE LISTVIEW

La classe *android.widget.ListView* permette di realizzare delle liste di elementi con scrolling verticale. Le istanze di *ListView*, come quelle di ogni altro widget, possono essere costruite indifferentemente con codice Java o XML. Nel primo caso, dall'interno di un'attività, si farà qualcosa come:

```
ListView listView = new ListView(this);
```

Nel caso del codice XML, invece, sarà necessario fare qualcosa del tipo:

```
<ListView
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:id="@+id/miaLista" />
```

Non dimenticatevi mai di assegnare un id ai vostri oggetti *ListView* dichiarati in un layout XML, in modo da poterli poi recuperare e sfruttare nel codice Java dell'attività con un'istruzione del tipo:

```
ListView listView = (ListView)
                findViewById(R.id.miaLista);
```

Una volta creato l'oggetto, bisogna assegnargli un *Adapter* affinché sia possibile caricare dei dati al suo interno. Il metodo utile è *setAdapter()*:

```
listView.setAdapter(mioAdapter);
```

Che adattatore utilizzare? Un *ArrayAdapter* o un adattatore custom ottenuto per estensione di *BaseAdapter*, come mostrato in precedenza, andranno benissimo.
Gli eventi di tocco e tocco lungo su un oggetto della lista possono essere gestiti applicando degli appositi listener all'oggetto *ListView*. Il tocco semplice può essere intercettato usando il metodo *setOnItemClickListener()* e l'interfaccia *android.widget.AdapterView.OnItemClickListener*:

```
listView.setOnItemClickListener(new
                        OnItemClickListener() {
    @Override
    public void onItemClick(AdapterView<?>
        adapterView, View view, int position, long id) {
        // ...
    }
});
```

Gli argomenti forniti al metodo *onItemClick()* sono:

- **AdapterView<?> adapterView**
  L'*AdapterView* che ha subito l'evento.
- **View view**
  Il widget all'interno dell'*AdapterView* che ha subito l'evento.
- **int position**
  La posizione del widget all'interno dell'*AdapterView*, come indice da zero in su.
- **long id**
  L'id della riga dell'*AdapterView* che ha subito l'evento.

Il tocco lungo richiede il metodo *setOnItem LongClickListener()* e l'interfaccia *android.wid-*

get.AdapterView.OnItemLongClickListener. Il metodo da implementare è *onItemLongClick()*. Gli argomenti sono gli stessi di *onItemClick()*:

```
listView.setOnItemLongClickListener(new
                                    OnItemLongClickListener()
{
    @Override
    public boolean onItemLongClick(AdapterView<?>
            adapterView, View view, int position, long id)
{
        // ...
    }
});
```

Nel CD-Rom allegato alla rivista trovate una demo riassuntiva che mette insieme un contenitore *ListView* con un adattatore *ArrayAdapter*.

LA CLASSE GRIDVIEW

La classe *android.widget.GridView* è il componente per costruire una tabella. Quando si crea l'oggetto è importante specificare quante colonne si vogliono usare al suo interno. In Java si fa così:

```
GridView gridView = new GridView(this);
gridView.setNumColumns(3);
```

In XML, invece:

```
<GridView android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    android:numColumns="3"
    android:id="@+id/miaGriglia" />
```

Altri metodi ed attributi permettono di impostare ulteriori proprietà della griglia:

- **setColumnWidth(int)** Imposta la larghezza delle colonne con un valore in pixel. L'attributo XML corrispondente è ***android:columnWidth***, che permette di specificare anche un'unità di misura differente dai pixel (ad esempio "*10pt*").

- **setHorizontalSpacing(int)** Imposta la distanza orizzontale tra un elemento e l'altro della griglia con un valore in pixel. L'attributo XML corrispondente è ***android:horizontalSpacing***, che permette di specificare anche un'unità di misura differente dai pixel per la dimensione.

- **setVerticalSpacing(int)** Imposta la distanza verticale tra un elemento e l'altro della griglia con un valore in pixel. L'attributo XML corrispondente è ***android:verticalSpacing***, che permette di specificare anche un'unità di misura differente dai pixel per la dimensione.

- **setGravity(int)** Imposta la costante di gravità per l'allineamento dei widget. Il valore di default è *Gravity.LEFT*. In XML l'attributo corrispondente è ***android:gravity***.

- **setStretchMode(int)** Quando la tabella è più piccola del contenitore al quale si deve adattare, questo metodo permette di specificare la politica da adottare per distribuire lo spazio in più fra le colonne. I valori possibili sono *GridView.NO_STRETCH* (nessun ridimensionamento automatico delle colonne rispetto alle loro dimensioni previste), *GridView.STRETCH_SPACING* (distribuisce lo spazio aggiuntivo come spaziatura tra una colonna e la successiva), *GridView.STRETCH_SPACING_UNIFORM* (come il precedente, ma aggiunge spazio anche prima della prima colonna e dopo l'ultima) e *GridView.STRETCH_ COLUMN_WIDTH* (assegna lo spazio aggiuntivo alle singole colonne). In XML l'attributo corrispondente è ***android:stretchMode***, ed i valori possibili sono *none*, *spacingWidth*, *spacingWidth Uniform* e *columnWidth*.

L'adapter può essere impostato chiamando il metodo *setAdapter()* su un oggetto *GridView*, proprio come nel caso di *ListView*. L'adattatore viene interrogato per estrarre gli elementi che faranno parte della griglia. Facciamo il caso che la griglia abbia tre colonne. Il primo elemento fornito dall'adattatore andrà alla posizione in riga 1 e colonna 1 della griglia; il secondo finirà a riga 1 e colonna 2, il terzo a riga 1 e colonna 3. Terminata la riga, il quarto elemento dell'adattatore proseguirà a partire dalla riga successiva. Quindi il quarto elemento sarà visualizzato a riga 2 e colonna 1, il quinto a riga 2 e colonna 2, e così via. La griglia, per farla più semplice, viene popolata da sinistra verso destra e dall'alto verso il basso. Anche in questo caso è possibile intercettare eventi di tocco e tocco lungo sui singoli elementi della griglia. Ancora una volta, i metodi utili sono *setOnItemClickListener()* e *setOnItemLongClickListener()*, con le corrispettive interfacce *AdapterView.OnItemClickListener* e *AdapterView.OnItemLongClickListener*. Nel CD-Rom troverete il codice completo dell'esempio mostrato in Fig. 4.

SPINNER

Uno spinner è un elenco a tendina dal quale è possibile selezionare un elemento. In Android gli spinner sono delle estensioni di *AdapterView*, e quindi rientrano nella panoramica odierna. Utilizzarli è davvero molto semplice, anche perché sono dei componenti più basilari rispetto ai *ListView* o i *GridView* esaminati nei paragrafi precedenti. Il

LISTACTIVITY

Quando un'attività è costituita esclusivamente da una lista, Android mette a disposizione una pratica e proficua scorciatoia: invece di creare una classe *Activity* all'interno della quale si deve definire un layout basato su un componente *ListView* ed il relativo *Adapter*, si può estendere direttamente la classe *android.app.ListActivity*. Così facendo non c'è bisogno di definire il layout, e l'adattatore può essere impostato direttamente sull'attività chiamando il suo metodo *setListAdapter()*.

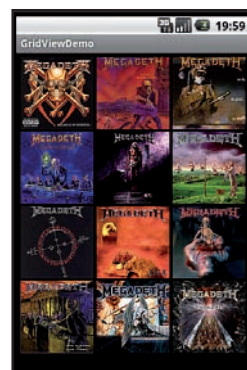



***Fig. 3:** GridView in azione, con una matrice di dodici immagini distribuite su tre colonne*

codice Java è il seguente:

```
Spinner spinner = new Spinner(this);
```

In XML diventa:

```
<Spinner android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:id="@+id/mioSpinner"
/>
```

Opzionalmente, con il metodo *setPrompt()* o con l'attributo *android:prompt*, è possibile dare un titolo alla lista di scelte che viene mostrata all'attivazione dello spinner. Per esempio:

```
spinner.setPrompt("Seleziona l'elemento che
                                    preferisci");
```

I dati mostrati nella lista saranno, naturalmente, forniti da un adattatore che potete implementare come meglio credete, per poi impostarlo con *setAdapter()*:

```
spinner.setAdapter(mioAdapter);
```

Fig. 5: Esempio di Spinner con elementi testuali

In molti casi gli spinner vengono utilizzati con elementi testuali, e quindi in accoppiata con un *ArrayAdapter*.

Tuttavia non esistono vincoli in merito: se nella vostra applicazione ha senso, potete utilizzare un qualsiasi altro adapter, per mostrare widget complessi, magari con immagini, come elementi della lista di selezione. Quale sia l'elemento selezionato è sempre possibile saperlo chiamando il metodo *getSelectedItem()*. Lo vediamo in questo esempio:

```
String voceSelezionata = (String)
                          spinner.getSelectedItem();
```

Su uno spinner non si possono usare *OnItemClickListener* o *OnItemLongClickListener*, come nei due componenti studiati in precedenza. Per intercettare la selezione di un elemento è però possibile fare ricorso al metodo *setOnItemSelectedListener()* e alla corrispettiva interfaccia *android.widget.AdapterView.OnItemSelectedListener*. Due, in questo caso, sono i metodi da implementare:

- ***onItemSelected(AdapterView<?> adapterView, View view, int position, long id)*** Questo metodo viene richiamato alla selezione di un elemento della lista. Gli argomenti sono equivalenti a quelli già conosciuti con i metodi *onItemClick()* di *OnItemClickListener*.
- ***onNothingSelected(AdapterView<?> adapterView)*** Questo metodo viene richiamato quando la selezione precedente viene annullata, e quindi nessuna voce è selezionata. L'argomento *adapterView* rappresenta l'oggetto che ha subito l'evento, che nel caso specifico sarà quindi uno *Spinner*.

Ecco un esempio in grado di funzionare con uno spinner i cui elementi sono stringhe:

```
spinner.setOnItemSelectedListener(new
                            OnItemSelectedListener()
  {...
```

Carlo Pelliccia lavora presso 4IT (www.4it.it), dove si occupa di analisi e sviluppo software per piattaforme Java. Nella sua carriera di technical writer ha pubblicato cinque manuali ed oltre centocinquanta articoli, molti dei quali proprio tra le pagine di ioProgrammo. Il suo sito, che ospita anche diversi progetti Java Open Source, è disponibile all'indirizzo www.sauronsoftware.it

IL COMPONENTE GALLERY

Chiudiamo la panoramica sugli *AdapterView* parlando del componente *android.widget.Gallery*. Come il nome lascia facilmente indovinare, si tratta di un widget particolarmente adatto per la costruzione di gallerie di immagini. L'oggetto *Gallery*, ad ogni modo, può gestire qualsiasi *View* al suo interno, e non solo quelle di tipo *ImageView*. Il risultato prodotto è quello di una galleria sfogliabile, dove i dati inseriti all'interno del contenitore possono essere spostati verso destra o verso sinistra con il tocco del dito, per visualizzare l'elemento successivo o precedente. L'impiego di *Gallery* ricalca il modello ripetuto più volte nei paragrafi precedenti. Si può cioè usare il codice Java:

```
Gallery gallery = new Gallery(this);
```

Oppure, se preferite, potete usare la formulazione XML:

```
<Gallery

    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    android:id="@+id/miaGalleria" />
```

Siccome spesso i componenti *Gallery* sono adoperati per sfogliare una serie di immagini, come adattatore potete usare l'*ImageAdapter* sviluppato qualche paragrafo sopra come estensione di *BaseAdapter*:

```
gallery.setAdapter(new ImageAdapter(this, images));
```

I listener utilizzabili sono, ancora una volta, *OnItemClickListener* e *OnItemLongClickListener*. Un esempio completo e funzionante lo potete trovare nel CD-Rom allegato alla rivista.

Carlo Pelliccia

UN'APPLICAZIONE CON STILE

IL DESIGN È UNO DEI FATTORI PIÙ IMPORTANTI IN AMBITO MOBILE. NON È SUFFICIENTE CHE UN'APPLICAZIONE FUNZIONI: DEVE ANCHE ESSERE ELEGANTE E GRADEVOLE ALLA VISTA. PER QUESTO OGGI SCOPRIREMO COME GESTIRE IL LOOK DELLE APPLICAZIONI

Per tutta la durata di questo corso, abbiamo più volte rimarcato come Android spicchi tra gli altri sistemi mobili per la modernità dei suoi concetti e per la tipologia dei suoi strumenti, soprattutto per quel che riguarda il design delle interfacce utente. Realizzare una UI per Android, infatti, è un'operazione che ricorda più il design di una pagina Web che non la costruzione di un'applicazione a finestre su un sistema desktop. Grazie al linguaggio di layout basato su XML, in Android ogni widget dell'interfaccia può essere velocemente espresso e configurato. Tra le tante cose che Android permette, c'è anche la possibilità di intervenire sull'aspetto di ciascun widget, modificandone ad esempio il colore, i bordi, lo stile del testo, l'immagine di sfondo e così via. Oggi ci concentreremo proprio su questo aspetto, introducendo i concetti di *stile* e *tema*.

BISOGNA AVERE STILE!

Nel gergo di Android (e non solo), uno stile è un insieme di proprietà che possono essere applicate ad un widget per modificarne l'aspetto esteriore. Ci è già capitato di utilizzare delle proprietà di stile in alcuni dei codici studiati durante gli appuntamenti precedenti. Per andare dritti al punto, prendiamo in esempio il caso di un componente *TextView* così definito:

```
<TextView
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:text="@string/message"
    android:textStyle="bold|italic"
    android:textColor="#FFFF00"
    android:textSize="20sp"
    android:typeface="monospace"
    android:padding="10dp" />
```

Questa etichetta di testo ricorre a diverse proprietà di stile. Nello specifico, si richiede esplicitamente che il testo sia grassetto e corsivo (con la proprietà *textStyle*), di colore giallo (*textColor*), grande 20 punti (*textSize*), con un font a spaziatura fissa (*typeface*) e distanziato di 10 punti dai bordi del widget (*padding*). Tutte queste proprietà potrebbero essere espresse anche con del codice Java, ma naturalmente in XML è molto più semplice configurare lo stile di un widget. Utilizzando Eclipse, poi, lo è ancora di più, visto che si possono utilizzare le procedure guidate messe a disposizione dal plug-in per lo sviluppo Android.

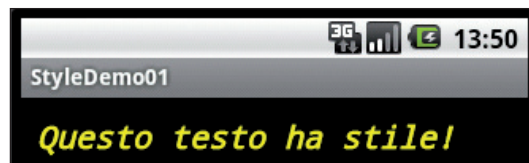



Fig. 1: Un componente TextView personalizzato nelle sue proprietà di stile

Supponiamo ora di voler sviluppare un'applicazione che faccia uso di diverse decine di componenti *TextView* come quello appena dimostrato. A livello di XML, magari aiutandoci con un po' di copia-incolla, potremmo naturalmente definire tanti widget tutti uguali, semplicemente replicando su ognuno di essi le medesime proprietà di stile. Come soluzione funziona, però non è il massimo della convenienza. Anzitutto ci sono un sacco di definizioni duplicate, e questo già di per sé non è un buon inizio. La cosa peggiore, però, è che diventa più difficile mantenere omogeneo lo stile di tutta l'applicazione. Facciamo ad esempio il caso che il cliente che ci ha commissionato l'applicazione, dopo averla vista, ci dice "mi sa che ho cambiato idea e che le scritte adesso le voglio verdi e un po' più grosse". In questo caso dovremmo tornare a lavorare su tutti gli XML, ricercando ogni occorrenza di *TextView* e andando a correggere le proprietà. Poi bisognerebbe riverificare ciascuna schermata per essere sicuri di non aver commesso errori. Insomma, in fin dei conti quel copia-incolla iniziale

potrebbe costarci parecchia fatica in futuro.
Per questo motivo Android mette a disposizione uno speciale costrutto XML che permette di definire gli stili come delle entità indipendenti, slegate cioè dal widget o dai widget cui sono applicate. Gli stili possono essere definiti negli XML di tipo *resources*, come già siamo abituati a fare con le stringhe. La cartella di progetto da utilizzare, quindi, è “res/values” (o una delle sue varianti) e il modello da seguire è il seguente:

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
  <style name="textViewStyle01">
    <item name="android:textStyle">bold|italic</item>

    <item name="android:textColor">#FFFF00</item>
    <item name="android:textSize">20sp</item>
    <item name="android:typeface">monospace</item>
    <item name="android:padding">10dp</item>
  </style>
</resources>
```

I singoli *<item>* rappresentano le proprietà che entrano a far parte dello stile. Nomi e valori, come è possibile osservare, devono essere ricavati dagli attributi dei widget e dai valori che è possibile attribuire a questi ultimi.
In un singolo file di risorse, naturalmente, possono essere definiti più stili, usando più occorrenze del tag *<style>*. Ciascuno stile deve avere un nome univoco. Il nome possiamo stabilirlo noi come meglio preferiamo, certamente facendo in modo che sia significativo nel nostro caso specifico. Nel caso appena mostrato, ad esempio, si sta preparando uno stile che si intende applicare a dei widget *TextView*. Per questo si è scelto di chiamarlo *textViewStyle01*. Il nome potrà successivamente essere impiegato per richiamare lo stile. Il modello da seguire in Java sarà:

```
R.style.textViewStyle01
```

Mentre in XML sarà:

```
@style/textViewStyle01
```

Una volta pronto, lo stile può essere applicato ad un widget qualsiasi attraverso il suo attributo *style*. Ad esempio:

```
<TextView
  android:layout_width="fill_parent"
  android:layout_height="wrap_content"
  android:text="@string/message"
  style="@style/textViewStyle01" />
```

Questo *TextView*, una volta caricato in un’attività, apparirà identico a quello dimostrato in apertura di paragrafo. Adesso, però, è possibile realizzarne a iosa senza dover fare copia-incolla degli attributi di stile: basterà applicare a tutti lo stile *textViewStyle01*, e tutti i testi appariranno gialli e in grassetto.
La richiesta del cliente di cambiare le dimensioni ed il colore del testo, adesso, potranno essere soddisfatte nel giro di un minuto, modificando un solo file XML, cioè quello che contiene la definizione dello stile.

EREDITÀ NEGLI STILI

Chi di voi conosce il Web design, l’HTML ed i CSS si starà sicuramente trovando a proprio agio. Il modello di gestione degli stili di Android, infatti, ha fatto tesoro dell’esperienza del Web. Dai fogli di stile CSS, infatti, riprende anche un’altra caratteristica: l’ereditarietà. Cerchiamo di capire insieme cosa significhi e quali vantaggi comporti.
Torniamo all’esempio del cliente petulante del paragrafo precedente. Nell’applicazione che stiamo realizzando per lui abbiamo fatto uso di numerosi oggetti *TextView*. Stando alla specifica iniziale del software, il testo in questi componenti deve apparire giallo e di una certa dimensione. Così abbiamo definito uno stile e lo abbiamo applicato ad ogni occorrenza del widget *TextView*. Il cliente, successivamente, ci ha chiesto delle modifiche, che noi siamo stati in grado di apportare istantaneamente modificando lo stile applicato al testo. La soddisfazione del cliente è stata enorme nel constatare quanto fossimo veloci ed efficienti nell’applicare la modifica richiesta. Per questo, il cliente ci ha preso gusto... Girovagando tra le schermate dell’applicazione, adesso gli è venuto in mente che alcune delle scritte che abbiamo modificato – solo alcune, però, non tutte – dovrebbero essere un po’ più piccole, ed inoltre andrebbero allineate al centro dello schermo invece che a sinistra. Contiamo fino a dieci, sfoggiamo un sorriso compiacente, annuiamo, e mettiamoci a lavoro.
Abbiamo diverse possibilità. Naturalmente non possiamo modificare lo stile *textViewStyle01*, perché questo significherebbe modificare tutti i *TextView* che ne fanno uso, e non solo quelle poche unità indicate dal cliente. Potremmo allora raggiungere ogni occorrenza di *TextView* da modificare, sganciarla dallo stile definito in precedenza, e definire su ciascuna di esse il nuovo stile usando gli attributi di stile previsti da *TextView*. Come soluzione funzionerebbe, ma sarebbe un passo indietro. La cosa migliore da fare, invece, è definire un secondo stile, da applicare poi in sostituzione del precedente soltanto a quelle etichette che devono essere modificate. La definizione degli stili, in questo caso,

FILE CONSIGLIATI

Stili e temi, come si è visto, vanno definiti in file XML di tipo *resources*, da posizionare al percorso */res/values* (o varianti). Non ci sono vincoli sui nomi dei file XML posizionabili a questo percorso, e pertanto potete crearne quanti ne volete e con i nomi che più desiderate. Potete mettere uno stile in ogni file, oppure fare un solo file con tutti gli stili della vostra applicazione. Per Android, in fin dei conti, è la stessa cosa. La maggior parte degli sviluppatori della comunità Android, ad ogni modo, preferisce usare un unico file *styles.xml* per gli stili ed un unico file *themes.xml* per i temi. Spesso risulta conveniente seguire questa convenzione.

diverrebbe:

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
  <style name="textViewStyle01">
    <item name="android:textStyle">bold|italic</item>
    <item name="android:textColor">#00FF00</item>
    <item name="android:textSize">20sp</item>
    <item name="android:typeface">monospace</item>
    <item name="android:padding">10dp</item>
  </style>
  <style name="textViewStyle02">
    <item name="android:textStyle">bold|italic</item>
    <item name="android:textColor">#00FF00</item>
    <item name="android:textSize">16sp</item>
    <item name="android:typeface">monospace</item>
    <item name="android:padding">10dp</item>
    <item name="android:gravity">center</item>
  </style>
</resources>
```

Adesso non dobbiamo far altro che andare a ricercare quelle *TextView* che devono usare il secondo stile al posto del primo, ed il gioco è fatto.

Fig. 2: Due componenti di testo con stili diversificati in alcune caratteristiche soltanto

NOTA

FULLSCREEN

Un'attività può essere mandata a tutto schermo applicandole il tema *Theme.NoTitleBar.Fullscreen.*

La soluzione funziona perfettamente, ma possiamo fare di meglio. Guardate attentamente la definizione dei due stili. C'è ridondanza di informazioni e, come sapete, ciò è male. I due stili definiti, infatti, sono correlati e per questo molto simili: differiscono solo in alcuni particolari. Il secondo stile introdotto, infatti, è solo una variazione del primo, e non un vero e proprio stile a sé. Se adesso il cliente ci venisse a chiedere di cambiare di nuovo il colore del testo, riportandolo a giallo, dovremmo modificare due stili anziché uno solo, e la cosa sarebbe innaturale proprio perché i due stili sono fortemente correlati. L'ereditarietà degli stili serve proprio per gestire situazioni di questo tipo:

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
  <style name="textViewStyle01">
    <item name="android:textStyle">bold|italic</item>
    <item name="android:textColor">#00FF00</item>
    <item name="android:textSize">20sp</item>
    <item name="android:typeface">monospace</item>
    <item name="android:padding">10dp</item>
  </style>
  <style name="textViewStyle01.PiccoloCentrato">
    <item name="android:textSize">16sp</item>
    <item name="android:gravity">center</item>
  </style>
</resources>
```

In questo caso non abbiamo più uno stile *textViewStyle02*. Abbiamo invece definito un sottostile di *textViewStyle01*, chiamato *textViewStyle01.PiccoloCentrato*. Questo nuovo stile è, come risultato visivo finale, identico al *textViewStyle02* del caso precedente, ma è concettualmente più corretto. Il solo fatto che sia un sottostile di *textViewStyle01*, fa sì che dallo stile genitore vengano ereditate tutte le proprietà precedentemente dichiarate. All'interno del sottostile, quindi, non è necessario specificare il colore, il padding, il tipo di carattere o qualsiasi altra proprietà in comune con il padre. All'interno del sottostile, invece, si devono specificare soltanto le differenze rispetto allo stile genitore, che in questo caso sono la dimensione e l'allineamento. Se il cliente ci dovesse chiedere di tornare ad utilizzare il colore giallo per le scritte, ci basterà modificare il valore della proprietà *android:textColor* di *textViewStyle01*, e la modifica sarà propagata automaticamente anche al sottostile *PiccoloCentrato*.
In XML ci si riferisce ad un sottostile usando la notazione puntata, così come si è fatto quando lo si è definito:

```
<TextView style="@style/textViewStyle01.PiccoloCentrato" ... />
```

In Java, invece, il punto diventa un trattino basso (*underscore*):

```
R.style.textViewStyle01_PiccoloCentrato
```

Un'altra cosa importante: non c'è un limite alla profondità dell'albero dei sottostili. Se adesso il cliente ci dovesse chiedere di fare in modo che alcune delle scritte piccole centrate abbiano anche un'ombra, potremmo definire un sotto-sottostile:

```
<style name="textViewStyle01.PiccoloCentrato.Ombra">
  <item name="android:shadowColor">#00FF66</item>
  <item name="android:shadowDx">1.5</item>
  <item name="android:shadowDy">1.5</item>
```

```
    <item name="android:shadowRadius">1.5</item>
</style>
```

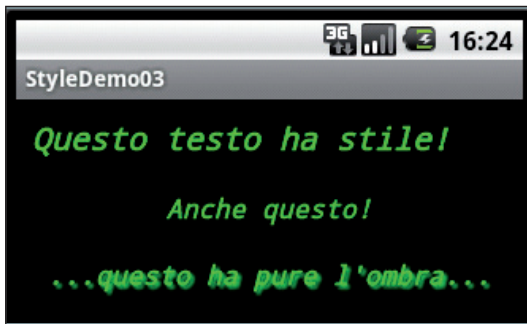



Fig. 3: Stili e sottostili in azione

TEMI

Quando si applica uno stile ad un widget, le proprietà dello stile specificato influenzano solo e soltanto il componente al quale sono state attribuite. Nessun altro widget sarà condizionato dallo stile applicato al componente. In Android esiste però la possibilità di applicare uno stile globale ad un'attività o ad un'applicazione, facendo in modo che venga automaticamente assorbito da tutti i widget mostrati. In questo caso si dice che si utilizza un *tema*.
Per impostare un tema per la vostra applicazione dovete per prima cosa creare uno stile. Prendiamo in considerazione il seguente:

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
  <style name="mioTema">
    <item name="android:background">#FFFFFF</
                                        item>
    <item name="android:textColor">#000000</item>
    <item name="android:textStyle">bold</item>
    <item name="android:textSize">16sp</item>
    <item name="android:typeface">monospace</
                                        item>
    <item name="android:padding">10dp</item>
    <item name="android:gravity">top|center</item>
  </style>
</resources>
```

Questo stile imposta uno sfondo bianco ed un testo nero, oltre a definire altre proprietà sul testo e sulla disposizione degli elementi. Per applicarlo all'intera applicazione, anziché ad un singolo elemento, dovete modificare *AndroidManifest.xml*. Là dove viene definita l'applicazione, fate come segue:

```
<application
  android:icon="@drawable/icon"
  android:label="@string/app_name"
  android:theme="@style/mioTema">
  ...
</application>
```

L'attributo *android:theme* può essere applicato anche al tag <*activity*>, se vi interessa fare in modo che attività differenti della stessa applicazione facciano uso di temi differenti.
Quando uno stile è applicato come tema, tutti i widget dell'applicazione o dell'attività ne ereditano le proprietà. Applicando il tema mostrato sopra, ad esempio, si otterrà un'applicazione dallo sfondo bianco e dalle scritte nere, senza la necessità di associare esplicitamente ai componenti utilizzati. Tutti i *TextView*, quindi, saranno automaticamente con sfondo bianco e testo nero.

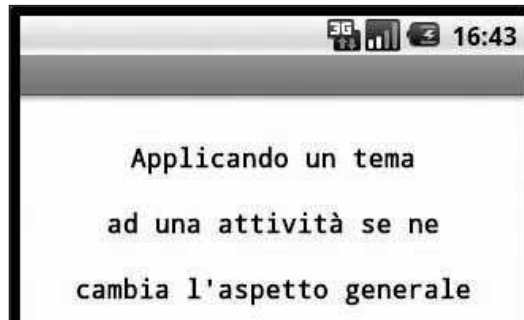



Fig. 4: Applicando un tema è possibile stravolgere completamente l'aspetto delle applicazioni

Non è detto che tutti i widget usati poi nell'applicazione supportino l'intero set di proprietà di stile specificate nel tema. Ad esempio, i widget *ImageView* non contengono testo e, di conseguenza, non supportano le proprietà che impostano colore, dimensione ed altri aspetti del testo. Quando ciò accade, il widget non fa altro che prelevare dal tema ed applicare soltanto quelle proprietà che è in grado di supportare.

STILI E TEMI PREDEFINITI

Quando non si applicano né stili né temi, all'interno delle nostre applicazioni, Android non fa altro che usare i suoi stili ed i suoi temi predefiniti. L'aspetto di ciascun widget, infatti, è definito attraverso un insieme di proprietà di stile che possiamo conoscere ed esplorare. Quando si applica uno stile personale ad un widget, in realtà, non si fa altro che sovrapporre il proprio stile a quello già predefinito per quel widget. Ma quanti e quali sono gli stili ed i temi previsti in Android? Non staremo qui ad elencarli uno ad uno, per motivi di spazio ma anche perché esistono differenze tra le diverse versioni del sistema. Potete, in base ad i vostri interessi, consultare quali siano gli stili ed i temi previsti per una specifica piattaforma Android.

DP E SP

Le unità di misura applicabili alle dimensioni, in Android, sono molteplici. Tuttavia le più consigliate sono *dp* e *sp*. La prima si applica alle distanze, ed è praticamente una misura in pixel riscalata in base alla densità dello schermo. Con schermi a 160 dpi (densità media) 1 dp corrisponde ad 1 px. Se la densità aumenta (ad esempio più di 200 dpi), però, la dimensione dei pixel diminuisce, e per questo il sistema fa in modo che 1 dp diventi 2 px o più. Viceversa avviene con densità inferiori alla media. Insomma, il *dp* è un'unità di misura che consente di preservare grosso modo le distanze al variare della densità dei pixel nello schermo. L'unità *sp* è simile, ma è pensata per essere applicata al testo: tiene infatti conto non solo della densità dello schermo, ma anche delle impostazioni dell'utente sulle dimensioni delle scritte. In un dispositivo impostato per visualizzare scritte molto grandi, 1 sp corrisponderà ad una dimensione maggiore rispetto ad un sistema con medesima densità di schermo ma impostato con caratteri più piccoli.

Nel disco rigido del computer che state utilizzando per sviluppare, localizzate dove avete installato l'SDK di Android. Qui dovreste avere una directory chiamata "*platforms*". Questa cartella contiene le diverse versioni di Android che avete scaricato ed integrato nel vostro SDK. Saranno cartelle del tipo "*android-1*", "*android-2*", "*android-3*" e così via, dove il numero associato alla cartella corrisponde alla revisione delle API abbinata (*Android 2.1*, ad esempio, è *API Level 7*). Scegliete la directory che corrisponde alla piattaforma di vostro interesse e seguite ora il percorso "*data/res/values*". Qui trovate i file di risorsa predefiniti per la piattaforma Android selezionata. Per stili e temi, rispettivamente, potete consultare i file *styles.xml* e *themes.xml*. Gli stili ed i temi predefiniti di Android possono essere riferiti in XML usando la sintassi:

```
@android:style/NomeStile
```

Il tema predefinito per le attività, ad esempio, è quello al percorso:

```
@android:style/Theme
```

Gli stessi stili e temi sono riferibili e consultabili anche da codice Java. Gli identificativi per ciascun stile sono disponibili nella speciale classe *R*, non quella di progetto, ma quella di sistema, che è riferibile con il percorso *android.R*. Il tema principale delle attività di Android, ad esempio, può essere riferito in Java così:

```
android.R.style.Theme
```

È interessante osservare come Android contenga delle varianti per ciascuno stile e per ogni tema predefinito. Ad esempio, il tema principale *Theme* prevede delle estensioni come *Theme.Light* (un tema dai colori chiari), *Theme.Translucent* (un tema

Fig. 5: Un'attività con l'aspetto di una finestra di dialogo, ottenuta applicando il tema predefinito Theme.Dialog

dallo sfondo trasparente), *Theme.Dialog* (il tema delle finestre di dialogo) ed altri ancora. Queste estensioni del tema principale sono lì proprio per essere adoperate da noi sviluppatori. Potremmo allora realizzare un'attività che appaia all'utente come una finestra di dialogo, facendo così nell'*AndroidManifest.xml*:

```
<activity android:theme="@android:style/Theme.
                                        Dialog">
```

ESTENDERE GLI STILI ED I TEMI PREDEFINITI

Gli stili ed i temi predefiniti possono anche essere estesi. Come abbiamo appena osservato, Android fornisce parecchie alternative ai temi e agli stili di base, ma naturalmente può capitare che neanche le alternative incorporate soddisfino i requisiti di una particolare applicazione. Ecco allora che ci ritroveremo a dover creare il nostro tema custom. Piuttosto che crearlo e definirlo da zero, che è un'operazione lunga e tediosa, possiamo pensare di estendere uno dei temi incorporati, variandolo solo là dove non ci sta bene. Uno stile o un tema predefinito possono essere estesi osservando la seguente sintassi XML:

```
<style name="mioStile" parent="@android:style/
                                  StilePredefinito">
  ...
</style>
```

Ad esempio:

```
<style name="mioTema" parent="@android:style/
                                        Theme">
  <item name="android:windowBackground">@drawa
                          ble/miobackground</item>
</style>
```

Si è appena realizzato un tema uguale in tutto e per tutto a quello di base, fatta eccezione per lo sfondo delle finestre, che sarà realizzato servendosi dell'immagine raggiungibile al percorso *@drawable/miobackground*. Non resta che applicare questo tema ad un'attività, come abbiamo imparato a fare nei paragrafi precedenti:

```
<activity android:theme="@style/mioTema">
```

Carlo Pelliccia

Fig. 6: Un tema costruito estendendo il tema di base di Android e modificando lo sfondo e le dimensioni del testo

Carlo Pelliccia lavora presso 4IT (www.4it.it), dove si occupa di analisi e sviluppo software per piattaforme Java. Nella sua carriera di technical writer ha pubblicato cinque manuali ed oltre duecento articoli, molti dei quali proprio tra le pagine di ioProgrammo. Il suo sito, che ospita anche diversi progetti Java Open Source, è disponibile all'indirizzo www.sauronsoftware.it

LO STORAGING SECONDO ANDROID

LEGGERE E SCRIVERE FILE DAL DISCO DI UNO SMARTPHONE ANDROID È UN'OPERAZIONE POSSIBILE MA SOGGETTA A RESTRIZIONI DI SICUREZZA E A NORME DI BUON USO. OGGI IMPAREREMO COME UTILIZZARE CORRETTAMENTE IL FILE SYSTEM DI ANDROID

Tempo di realizzazione

Conclusa la panoramica dedicata alla realizzazione delle UI (interfacce utente) delle applicazioni Android, andiamo da oggi a concentrarci su altri aspetti della piattaforma mobile di Google, egualmente indispensabili per applicazioni distribuibili e vendibili nel mondo reale. Cominciamo parlando di *file system*. Qualunque applicazione Android, infatti, può leggere e scrivere file dalla memoria interna del telefono o da una scheda esterna inserita nel dispositivo. I principi da osservare per compiere questo genere di operazioni differiscono leggermente dalle normali pratiche Java adoperate negli applicativi desktop o server. Scopriamo insieme come fare per leggere e scrivere file dall'interno di un'applicazione Android.

SICUREZZA PRIMA DI TUTTO

Le piattaforme mobili di nuova generazione sono figlie di una ritrovata attenzione per la sicurezza dell'utente, dei suoi dati e del suo dispositivo. I sistemi operativi per desktop, Windows in primis, sono continuamente vittima di virus e malware di ogni sorta. Questa piaga è da sempre causa di grossa frustrazione per chi il computer lo utilizza per svago o per lavoro, pur non essendone un esperto e non comprendendone i meccanismi interni. Siccome gli smartphone sono sempre più simili ai PC, sia come potenza di calcolo sia come possibilità dei loro software, i produttori di sistemi mobili stanno facendo il possibile per evitare che anche le loro piattaforme possano diventare vittima di tale piaga.
Il software malevolo può insediarsi in un sistema in due differenti maniere: sfruttando una vulnerabilità interna del sistema operativo o di qualche software che vi è installato, oppure convincendo l'utente ad eseguire un programma che all'apparenza è innocuo, ma che in realtà nasconde qualcosa di losco. Sul primo fronte si combatte una battaglia fatta di investimenti sulla sicurezza del codice prodotto. I creatori di Android e di tutti i sistemi operativi contemporanei, fortunatamente per noi, spendono molte risorse nella revisione del loro codice e nei test di sicurezza dello stesso. Significa che i sistemi operativi ed i software che girano al loro interno, oggi, sono molto meno vulnerabili rispetto a qualche anno fa. Naturalmente non esiste e non esisterà mai un sistema o un software sicuro al 100%, e per questo i produttori investono anche nel correggere velocemente le falle che vengono scoperte, distribuendo poi gli aggiornamenti in maniera rapida e trasparente.
Sul secondo fronte, quello cioè che fa leva sul fattore umano, la battaglia è invece un po' più complessa ed arretrata. C'è chi, come Apple, richiede di approvare preventivamente qualunque applicazione prodotta per le piattaforme iPhone, iPod e iPad. L'utente può installare applicazioni solo se le preleva dallo store di Apple, che ha verificato una ad una le applicazioni disponibili, convalidandone sicurezza, attendibilità e funzionalità. L'utente, così, non rischia di cadere in trappola. Da una parte questo approccio risolve quasi completamente il problema del fattore umano della sicurezza, ma dall'altro limita pesantemente la libertà dell'utente e dei produttori di software. Insomma, la cura rischia di essere peggiore del male. Android, al contrario, è permeato da una filosofia più aperta, e per questo lascia maggiore libertà ad utenti e programmatori, senza costringerli ad un unico canale di approvvigionamento delle applicazioni. Tutto ciò, però, deve essere conciliato con la necessità di non far proliferare il malware su questa piattaforma software. Nel caso di Android, quindi, non esiste un unico store delle applicazioni, ma ne esistono diversi. C'è quello principale, l'*Android Market* di Google, ma per il resto chiunque è libero di realizzare il proprio, ed infatti diversi produttori lo hanno già fatto. In questo caso, quindi, si può scegliere di chi fidarsi, godendo sia di una maggiore libertà sia di un senso di sicurezza basato sulla fiducia per lo store che si sta adoperando. Ad ogni modo resta sempre viva la possibilità di installare applicazioni procurate anche senza la mediazione di uno store. È dunque necessario che il sistema sia intrinsecamente sicuro, in modo che l'utente corra meno rischi possibile.

STORAGE INTERNO

Le applicazioni Android dispongono tutte di una porzione di spazio sul file system all'interno del quale possono creare e leggere dei file. Tale spazio è appannaggio esclusivo dell'applicazione: altri pacchetti installati nel dispositivo non possono farvi accesso. Insomma, ciascuna applicazione dispone di un'area protetta ed esclusiva all'interno della quale può fare ciò che vuole, senza però arrecare disturbo al sistema o alle altre applicazioni che vi sono installate. La classe *android.content.Context*, che è quella da cui derivano *Activity* e tutti gli altri mattoni fondamentali di Android, dispone di una serie di metodi utili per interagire con la porzioni di file system esclusivamente assegnata all'applicazione. Per scrivere un file all'interno dell'area è disponibile il metodo:

public FileOutputStream openFileOutput(String name, int mode)
throws FileNotFoundException

Il parametro *name* è il nome del file da scrivere, mentre il parametro *mode* può essere:

- ***Context.MODE_PRIVATE***
  Rende il file privato, cioè appannaggio esclusivo dell'applicazione che lo sta scrivendo. Nessun'altra applicazione potrà vederlo, leggerlo o sovrascriverlo.
- ***Context.MODE_APPEND***
  Agisce in *append* sul file specificato, cioè se il file già esiste, invece di sovrascriverlo, gli accoda i nuovi byte che saranno scritti nello stream. Utile quando si generano report e log.
- ***Context.MODE_WORLD_READABLE***
  Rende il file accessibile in sola lettura dalle altre applicazioni installate nel sistema.
- ***Context.MODE_WORLD_WRITEABLE***
  Rende il file accessibile in sola scrittura dalle altre applicazioni installate nel sistema.

Due o più costanti possono essere applicate contemporaneamente con l'operatore binario *OR* (simbolo: *pipe*). Ad esempio se si vuole generare un file privato in append si può fare:

Context.PRIVATE | Context.APPEND

Un file condiviso con le altre applicazioni sia in lettura che in scrittura, invece, dovrà avere modo:

Context.MODE_WORLD_READABLE | Context.MODE_WORLD_WRITEABLE

Il metodo *openFileOutput()* restituisce un oggetto *java.io.FileOuputStream*, che può essere pertanto manipolato come un qualsiasi output stream di Java. Si faccia pertanto riferimento alla documentazione Java per quel che riguarda l'utilizzo di stream e affini. Infine *openFileOutput()* può propagare una *java.io.FileNotFoundException*. Ciò avviene quando il file non può essere creato perché non valido.
Il file creati con *openFileOutput()* possono successivamente essere riletti servendosi di un altro metodo messo a disposizione da *Context*:

public abstract FileInputStream openFileInput (String name)
throws FileNotFoundException

Il parametro *name*, come è facile immaginare, è il nome del file da recuperare. L'eccezione *java.io.FileNotFoundException* viene propagata se il file richiesto non esiste. L'oggetto restituito è un input stream standard di tipo *java.io.FileInputStream*. Questo oggetto, come nel caso precedente, può essere utilizzato secondo la comune prassi Java per la lettura dei contenuti del file. I file non più utili possono essere cancellati con il metodo:

public boolean deleteFile(String name)

In questo caso non ci sono eccezioni da gestire, ma il metodo restituisce un booleano per indicare se il file specificato è stato effettivamente rimosso oppure no.
Infine i file conservati nell'area riservata all'applicazione possono essere elencati con il metodo:

public String[] fileList()

Il metodo restituisce un array con i nomi di tutti i file associati all'applicazione.

REALIZZIAMO UN BLOCCO NOTE

Realizziamo insieme un'applicazione dimostrativa in grado di scrivere e leggere da un file conservato nello spazio riservato all'applicazione stessa. Realizzeremo una specie di blocco note, che l'utente potrà utilizzare per prendere appunti. Gli appunti saranno salvati su un file interno all'applicazione, che l'utente potrà successivamente richiamare. Chiameremo l'applicazione ed il corrispondente progetto "*FileDemo01*". Partiamo definendo le seguenti risorse su *res/values/strings.xml*:

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <string name="app_name">FileDemo01</string>
    <string name="saveButton">Salva</string>
    <string name="loadButton">Carica</string>
</resources>
```

ANDROID 2.2
Se siete appassionati di Android, certamente la notizia non vi sarà sfuggita: è stato da poco rilasciato Android 2.2, nome in codice **Froyo**. Le principali novità di questa release riguardano le presentazione della macchina virtuale Dalvik: grazie all'introduzione della compilazione JIT (Just In Time) il codice Java riesce a correre fino al 400% più velocemente! Dal punto di vista di noi sviluppatori, ci sono anche novità a livello delle interfacce di programmazione, che hanno raggiunto l'API Level 8. Simultaneamente è stata rilasciata anche la versione R6 dell'SDK e la 0.9.7 del plug-in ADT per Eclipse. Aggiornate pertanto tutti i vostri ambienti alle nuove versioni.

A queste affianchiamo il seguente layout da posizionare su *res/layout/main.xml*:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.
                                          com/apk/res/android"
  android:layout_width="fill_parent"
  android:layout_height="fill_parent"
  android:orientation="vertical">
  <LinearLayout android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:orientation="horizontal">
    <Button android:layout_width="wrap_content"
      android:layout_height="wrap_content"
      android:id="@+id/saveButton"
      android:text="@string/saveButton" />
    <Button android:layout_width="wrap_content"
      android:layout_height="wrap_content"
      android:id="@+id/loadButton"
      android:text="@string/loadButton" />
  </LinearLayout>
  <EditText android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    android:id="@+id/textArea"
    android:inputType="textMultiLine"
    android:gravity="top|left" />
</LinearLayout>
```

Questo layout riempie il display con una casella di testo, all'interno della quale l'utente potrà appuntare le proprie note. Al di sopra di essa sono stati disposti i due pulsanti "*Salva*" e "*Carica*", utili rispettivamente per memorizzare e per richiamare successivamente il testo digitato. Realizziamo ora l'attività *it.ioprogrammo.filedemo01.FileDemo01Activity*, incaricata di realizzare la logica di scrittura e lettura del file su comando dell'utente:

```
package it.ioprogrammo.filedemo01;
...
public class FileDemo01Activity extends Activity {
  @Override
  public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.main);
    Button saveButton = (Button) findViewById
                                   (R.id.saveButton);
    saveButton.setOnClickListener(new View.
                                   OnClickListener() {
      @Override
      public void onClick(View v) {
        save("testo.txt");
      }
    });
```

```
    Button loadButton = (Button) findViewById
                                   (R.id.loadButton);
    loadButton.setOnClickListener(new View.w()
    {
      @Override
      public void onClick(View v) {
        load("testo.txt");
      }
    });
  }
  private void save(String filename) {
    EditText textArea = (EditText) findViewById
                                   (R.id.textArea);
    String text = textArea.getText().toString();
    Writer writer = null;
  ...
```

L'attività gestisce il file riservato *testo.txt* attraverso i due metodi *load()* e *save()*, richiamati alla pressione dei due bottoni disposti nel layout. Il testo viene letto e scritto servendosi delle astrazioni *Reader* e *Writer* di Java (cfr. box laterale), utili quando si ha a che fare con file di natura testuale.
Sul CD-Rom allegato alla rivista troverete sia l'esempio completo dell'applicazione "*FileDemo01*", sia una seconda applicazione "*FileDemo02*" che usa il metodo *fileList()* e delle finestre di dialogo per far consentire all'utente di scegliere il nome del file da salvare o da caricare.

STORAGE ESTERNO

I dispositivi Android possono disporre di un secondo spazio di storage, definito "storage esterno". Solitamente lo storage esterno è una scheda che può all'occorrenza essere rimossa e sostituita, ma non è detto: in alcuni casi lo storage esterno è comunque interno al dispositivo e non rimovibile. A priori, ad ogni modo, non è dato saperlo.
Pertanto la prima cosa da farsi quando si vuole accedere allo storage esterno, è controllare se questo è disponibile. Il metodo utile per farlo è contenuto staticamente nella classe *android.os.Environment*, ed è:

public static String getExternalStorageState()

La stringa restituita può essere confrontata con una delle seguenti costanti:

- ***Environment. MEDIA_MOUNTED***
  Lo storage esterno è disponibile e pronto.
- ***Environment. MEDIA_MOUNTED_READ_ONLY***
  Lo storage esterno è disponibile e pronto, ma è possibile accedervi in sola lettura.
- ***Environment. MEDIA_UNMOUNTED***
  Environment. MEDIA_UNMOUNTABLE
  Environment. MEDIA_BAD_REMOVAL
  Environment. MEDIA_CHECKING
  Environment. MEDIA_NOFS

NOTA

READER E WRITER

java.io.Reader e java.io.Writer sono le astrazioni di base di Java per la lettura e la scrittura dei testi. A differenza di *java.io.InputStream* e *java.io.OutputStream*, che ragionano in termini di byte, i reader ed i writer trattano caratteri, sequenze di caratteri e perciò stringhe. Passare da un *InputStream* ad un *Reader* è sempre possibile attraverso la classe ponte *java.io.InputStreamReader*. Per passare da un *OutputStream* ad un *Writer*, invece, ci vuole un *java.io.OutputStreamWriter*.

Environment. MEDIA_REMOVED
Environment. MEDIA_SHARED
Queste altre costanti rappresentano stati di errore per cui, per un motivo o per un altro, lo storage esterno non è disponibile. La documentazione ufficiale approfondisce ciascuno di questi stati.

Solitamente, prima di accedere allo storage esterno, si usa una routine del tipo:

```
boolean possoLeggereStorageEsterno = false;
boolean possoScrivereStorageEsterno = false;
String state = Environment.getExternalStorageState();
if (Environment.MEDIA_MOUNTED.equals(state)) {
  // Storage esterno disponibile in lettura e scrittura.
  possoLeggereStorageEsterno = possoScrivereStora
                                geEsterno = true;
} else if (Environment.MEDIA_MOUNTED_READ_ONLY.
                                equals(state)) {
  // Storage esterno disponibile solo in lettura.
  possoLeggereStorageEsterno = true;
  possoScrivereStorageEsterno = false;
} else {
  // Storage esterno non disponibile.
  possoLeggereStorageEsterno = possoScrivereStora
                                geEsterno = false;
}
```

Una volta che ci si è accertati che sia possibile accedere allo storage esterno, è possibile farlo recuperandone il percorso attraverso il metodo statico di *Environment*:

public static File getExternalStorageDirectory()

Il metodo restituisce un oggetto *java.io.File* che rappresenta la radice dello storage esterno. Usando le comuni API I/O di Java, a questo punto, è possibile navigare lo storage esterno, creare nuovi file, leggere quelli esistenti e così via, senza alcuna limitazione.
Sul CD trovate una rivisitazione del blocco note usato come caso di studio nel paragrafo precedente. Questa terza implementazione del software salva gli appunti dell'utente sullo storage esterno.

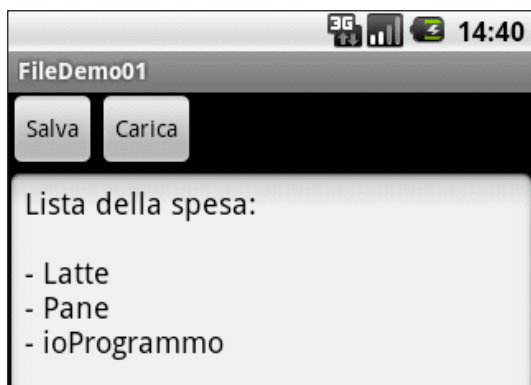



***Fig. 1:** L'applicazione "FileDemo01" realizza una specie di semplice blocco per gli appunti*

ORGANIZZAZIONE DELLO STORAGE ESTERNO

Quando si utilizza lo storage esterno, ci sono delle norme che è opportuno seguire. Ad esempio si sconsiglia di creare file direttamente nella radice dello storage. Android infatti organizza il suo storage esterno con una serie di directory standard che, in molti casi, è conveniente utilizzare. Queste solitamente sono:

- *Music*, per la musica.
- *Podcasts*, per i podcast.
- *Ringtones*, per le suonerie.
- *Alarms*, per i suoni da abbinare agli allarmi.
- *Notifications*, per i suoni da abbinare alle notifiche.
- *Pictures*, per le foto (escluse quelle fatte con la fotocamera del dispositivo).
- *Movies*, per i video (esclusi quelli ripresi con la videocamera del dispositivo).
- *Download*, per i file scaricati.

Seguendo questa convenzione diventa molto semplice condividere dati con le altre applicazioni installate nel sistema. Ad esempio è possibile realizzare un'attività in grado di mostrare le immagini memorizzate nella card esterna, alla seguente maniera:

```
package it.ioprogrammo.filedemo04;
...
public class FileDemo04Activity extends Activity {

  public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.main);
    Drawable[] pictures;
    if (canReadFromExternalStorage()) {
      pictures = loadPicturesFromExternalStorage();
    } else {
      pictures = new Drawable[0];
    }
    ImageAdapter adapter = new ImageAdapter(this,
                                        pictures);
    Gallery gallery = (Gallery) findViewById(R.
                                    id.myGallery);
    gallery.setAdapter(adapter);
  }

  private boolean canReadFromExternalStorage() {
    String state = Environment.getExternalStorageState();
    if (Environment.MEDIA_MOUNTED.equals(state)) {
      return true;
  ...
```

L'esempio completo lo trovate nel CD-Rom. Nel prossimo numero impareremo a mettere un database all'interno delle nostre applicazioni Android!

Carlo Pelliccia

DOVE SI TROVA LO STORAGE DELLA MIA APP?

È possibile scoprire quale sia la directory radice dello spazio riservato ad un'applicazione, chiamando il metodo di *Context getFilesDir()*. Il metodo restituisce un oggetto di tipo *java.io.File*. Stampando il percorso della directory chiamandone il metodo di *File getAbsolutePath()*, scoprirete un risultato del tipo:

/data/data/<package applicazione>/files

Con la prospettiva *DDMS* di Eclipse, inoltre, potrete esplorare il file system dell'emulatore o di un dispositivo collegato, andando così a verificare dove sono i file e di quale permessi dispongono.

Carlo Pelliccia lavora presso 4IT (www.4it.it), dove si occupa di analisi e sviluppo software per piattaforme Java. Nella sua carriera di technical writer ha pubblicato cinque manuali ed oltre duecento articoli, molti dei quali proprio tra le pagine di ioProgrammo. Il suo sito, che ospita anche diversi progetti Java Open Source, è disponibile all'indirizzo www.sauronsoftware.it

DATABASE DA TASCHINO

UNA DELLE CARATTERISTICHE PIÙ INTERESSANTI DI ANDROID È IL DBMS INTEGRATO NEL SISTEMA, CHE DOTA LE APPLICAZIONI DELLA CAPACITÀ DI ARCHIVIARE E RICERCARE VELOCEMENTE I DATI. IN QUESTO ARTICOLO IMPAREREMO COME APPROFITTARNE

DBMS, ossia *Database Management System*, è un termine caro agli sviluppatori di applicazioni server-side e di impresa. La maggior parte delle applicazioni Web, ad esempio, si appoggiano ad un database, e lo utilizzano per organizzare e ricercare velocemente dati di ogni tipo. È naturale che le applicazioni server-side abbiano bisogno di un DBMS, visto che devono gestire grosse moli di dati, come l'elenco degli utenti iscritti ad un sito, o gli articoli pubblicati in un blog, i commenti dei visitatori, le statistiche di accesso e così via. Insomma, anche le più semplici applicazioni server fanno solitamente uso intensivo dei database.
Le applicazioni client-side per desktop, diversamente, usano i DBMS assai meno di frequente. Esistono dei DBMS specifici per personal computer, come Access di Microsoft, ed alcuni di quelli usati lato server si comportano bene anche se installati su un desktop o un laptop (MySQL, ad esempio). In questo caso i DBMS vengono utilizzati soprattutto in applicazioni specifiche, spesso realizzate ad hoc per particolari situazioni ed utenti. Ad esempio un gestionale per un libero professionista, o qualche altro software del genere, può fare ricorso ad un DBMS locale.
Passando al mondo mobile, invece, "DBMS" e "database" sono parole dal suono alieno e fuori contesto. Almeno fino a ieri.
Gli smartphone di nuova generazione, infatti, hanno raggiunto potenza di calcolo e complessità software sufficienti per l'esecuzione di un DBMS di piccole dimensioni, ottimizzato appositamente per un ambito peculiare e ristretto. Android comprende un piccolo DBMS, che gli sviluppatori possono richiamare per salvare ed organizzare i dati delle loro applicazioni. In questo articolo scopriremo come funziona questo DBMS. Vedremo come utilizzarne i benefici per rendere migliori e più complete le nostre applicazioni.

A COSA SERVE?

È lecito chiedersi che ruolo possa avere un DBMS installato in un telefono cellulare. Le applicazioni mobili, infatti, non gestiscono grandi quantità di dati, e nemmeno devono soddisfare il requisito dell'accesso contemporaneo da parte di un elevato numero di utenti. I DBMS, invece, servono proprio per soddisfare la necessità di scrivere e leggere molti dati, e soprattutto lo possono fare in tante sessioni simultanee (si pensi a quante visite contemporanee può raggiungere un sito Web). Un'applicazione mobile, da questo punto di vista, potrebbe tranquillamente accontentarsi di salvare le proprie informazioni in uno o più file di testo, come d'altronde è stato già spiegato nel numero precedente. Perché allora si dovrebbe usare un DBMS?
Tanto per cominciare, lo si fa perché con un DBMS è tutto più facile. In un database, infatti, i dati possono essere strutturati e tipizzati. Immaginiamo di dover realizzare un'applicazione che gestisca una lista di contatti, tipo un'agenda. Ogni contatto è caratterizzato da un identificativo *(ID)*, un nome, un cognome e un indirizzo. Se dovessimo salvare queste informazioni su un file di testo, dovremmo inventare un formato e programmare poi le classi in grado di gestirlo. La soluzione più classica consiste nell'organizzare un contatto per ciascuna riga, separando poi i campi con una virgola. Ad esempio:

1,Mario,Rossi,061299178
2,Antonio,Verdi,028667661

I dati di un contatto, inoltre, non sono soltanto di natura testuale: *nome, cognome*, e *numero di telefono* sono delle stringhe, ma l'identificativo del contatto è certamente un intero, e va gestito come tale. Nel leggere e nel salvare i dati, quindi, bisognerebbe fare un sacco di lavoro di codifica e decodifica, sia per gestire la struttura

del file sia per una corretta interpretazione dei tipi coinvolti.
Con un database, invece, è tutto più semplice: si crea una tabella con tanti campi quanti sono quelli necessari, ognuno già abbinato al tipo più consono. Le operazioni di lettura e scrittura, servendosi della libreria di accesso al DBMS, necessitano ora di pochissime righe di codice. Non è solo un fatto di fatica risparmiata, ma è anche una questione di sicurezza ed efficienza.
Un altro settore nel quale è difficile competere con un DBMS è quello delle ricerche: nell'agenda basata su file di testo bisognerebbe implementare degli algoritmi per la ricerca dei contatti. È possibile farlo, ma un buon algoritmo di ricerca, che allo stesso tempo sia tanto veloce quanto poco pretenzioso di risorse, non è una cosa semplice. Considerando che un DBMS contiene il risultato di anni di sviluppo nel settore delle ricerche ottimizzate, ecco che l'ago della bilancia si sposta ulteriormente dalla parte dei database.
Tutto ciò non significa, naturalmente, che qualsiasi applicazione Android debba usare il DBMS. La possibilità comunque c'è, e conviene sfruttarla in tutte quelle applicazioni che hanno bisogno di scrivere, leggere e fare ricerche su insiemi di dati strutturati.

SQLITE

Il DBMS integrato in Android arriva dal mondo dell'Open Source. Si tratta di *SQLite*, che nella costellazione dei DBMS si distingue per leggerezza e facilità di integrazione. Traducendo letteralmente quanto è possibile leggere nella home page del progetto: "SQLite è una libreria che implementa un motore di database SQL transazionale auto-sufficiente, senza server e senza bisogno di configurazione". SQLite è scritto in C e la sua distribuzione ufficiale funziona su Linux, Mac OS X e Windows. Per utilizzarlo su Android, comunque, non bisogna né compilare né installare nulla: è tutto compreso nel sistema operativo stesso. Gli sviluppatori di Google, come vedremo tra poco, hanno anche realizzato le interfacce di programmazione Java utili per richiamare ed utilizzare i database dall'interno di una qualsiasi applicazione Android. Insomma, leggere e scrivere un database, in Android, è un'operazione nativa e incorporata, come lo è accedere ad un file o connettersi alla rete. L'unico prerequisito richiesto è la conoscenza, anche basilare, di SQL, il linguaggio principe per l'interazione con i DBMS (cfr. box laterale).

RICHIAMARE UN DATABASE

Nella libreria di Android, i package Java di riferimento per l'accesso ai database e l'utilizzo di SQLite sono, rispettivamente, *android.database* e *android.database.sqlite*. I database, in Android, funzionano con la stessa logica di accesso e protezione dei file (cfr. numero precedente): ogni applicazione crea ed utilizza uno o più database in maniera esclusiva.
I database creati da un'applicazione non possono essere letti dalle altre applicazioni. La condivisione di dati strutturati fra più applicazioni Android, infatti, avviene mediante un ulteriore meccanismo, quello dei *provider*, che analizzeremo in futuro. Ciascuna applicazione, quindi, ha i suoi database, e nessun altro può accedervi.
La pratica di programmazione consigliata per l'accesso ad un database passa per l'estensione della classe *android.database.sqlite.SQLiteOpenHelper*. Il modello da seguire è il seguente:

```
package mia.applicazione;

import android.content.Context;
import android.database.sqlite.SQLiteDatabase;
import android.database.sqlite.SQLiteOpenHelper;

public class MioDatabaseHelper extends
SQLiteOpenHelper {

  private static final String DB_NAME = "nome_db";
  private static final int DB_VERSION = 1;

  public MioDatabaseHelper(Context context) {
    super(context, DB_NAME, null, DB_VERSION);
  }

  @Override
  public void onCreate(SQLiteDatabase db) {
    // Creazione delle tabelle
  }

  @Override
  public void onUpgrade(SQLiteDatabase db,
  int oldVersion, int newVersion) {
    // Aggiornamento delle tabelle
  }

}
```

Si deve creare una classe di questo tipo per ciascun database necessario alla propria applicazione. Nelle due costanti *DB_NAME* e *DB_VERSION* vanno indicati, rispettivamente, il nome del database (inutile dire che ogni data-

SQL

È impossibile avere a che fare con un DBMS senza masticare un po' di SQL, il linguaggio principe per l'utilizzo dei database. Questo linguaggio, almeno per quello che riguarda l'utilizzo quotidiano e non avanzato, è piuttosto semplice e consiste di pochi costrutti per l'interrogazione e la manipolazione dei dati. Insomma, se non ne sapete nulla non dovete disperare: l'SQL di base si impara facilmente ed in poco tempo. Cominciate con qualche tutorial online, come ad esempio questo: http://database.html.it/guide/leggi/40/guida-linguaggio-sql/

base di una stessa applicazione deve avere un nome differente, così come differente deve essere il nome della classe helper) e la sua versione. Il numero di versione è un intero che va incrementato ogni volta che, in un nuovo rilascio del software, il database viene modificato nella struttura. In questa maniera, quando l'utente aggiorna la propria applicazione, la classe riesce a capire se il database della versione precedentemente installata deve essere aggiornato oppure no.
I due metodi *onCreate()* e *onUpdate()* devono necessariamente essere implementati. Il primo viene richiamato quando il database deve essere creato per la prima volta. Il parametro *db*, di tipo *android.database.sqlite.SQLiteDatabase*, serve per poter manipolare il database appena creato. Tra poco vedremo come. Il metodo *onCreate()* viene richiamato quando l'applicazione è stata appena installata. Il metodo *onUpdate()*, invece, viene richiamato quando il database è già presente sul sistema, ma stando al numero di versione richiesta, risulta obsoleto. Di solito avviene dopo aver cambiato la versione di un'applicazione già installata.
I parametri *oldVersion* e *newVersion*, come è facile indovinare, indicano rispettivamente la versione già installata del database e quella ancora da installare.
Le tabelle di un database, in SQLite come in tutti i principali DBMS, debbono essere create con delle istruzioni SQL di tipo *CREATE TABLE*, oppure aggiornate con *ALTER TABLE*. Come appena detto, è necessario farlo all'interno dei metodi *onCreate()* e *onUpdate()*.
Gli oggetti di tipo *SQLiteDatabase* dispongono di un metodo *execSQL()*, che permette di eseguire del codice SQL arbitrario. È proprio quello che fa al caso nostro! Ad esempio:

```
@Override
public void onCreate(SQLiteDatabase db) {
   String sql = "";
   sql += "CREATE TABLE agenda (";
   sql += "  _id INTEGER PRIMARY KEY,";
   sql += "  nome TEXT NOT NULL,";
   sql += "  cognome TEXT NOT NULL,";
   sql += "  telefono TEXT NOT NULL";
   sql += ")";
   db.execSQL(sql);
}
```

In questo codice si è creata una tabella chiamata "agenda", con i campi citati nell'esempio usato in precedenza. Un trucco ed un consiglio: dotate sempre le vostre tabelle di una chiave primaria numerica. I campi di tipo *INTEGER PRIMARY KEY*, in SQLite, vengono sempre popolati automaticamente quando si aggiunge un nuovo record, con una sequenza progressiva di interi, funzionando quindi da contatore. Lo speciale nome *_id*, poi, serve ad Android per capire quale è il campo identificativo di ciascun record.
Seguendo questa prassi, le vostre tabelle saranno perfettamente integrate nella gestione automatica offerta dal sistema.
Una volta completato lo sviluppo della classe helper, è possibile sfruttarla in qualsiasi punto dell'applicazione. Tipicamente, nello sviluppo di una activity, si tende a conservare l'helper a livello di istanza, avendo cura di istanziarlo la prima volta in risposta all'evento *onCreate()* dell'attività. Il modello è il seguente:

```
package mia.applicazione;

import android.app.Activity;
import android.os.Bundle;

public class MiaAttivita extends Activity {

   private MioDatabaseHelper mioDatabaseHelper;

   @Override
   public void onCreate(Bundle savedInstanceState) {
     super.onCreate(savedInstanceState);
     mioDatabaseHelper = new MioDatabaseHelper(this);
   }

}
```

In qualsiasi punto dell'attività, adesso, è possibile utilizzare l'helper e domandargli l'accesso al database. Lo si può fare con i metodi *getReadableDatabase()* e *getWritableDatabase()*:

```
SQLiteDatabase db = mioDatabaseHelper.get
                                ReadableDatabase();
```

oppure

```
SQLiteDatabase db = mioDatabaseHelper.
                                getWritableDatabase();
```

Con *getReadableDatabase()* si ottiene l'accesso al database in sola lettura; con *getWritableDatabase()* si ottiene invece la possibilità di scrivere nel database, per inserire, aggiornare o cancellare record dalle tabelle. In ambo i casi l'oggetto restituito è di tipo *android.database.sqlite.SQLiteDatabase*. Nei prossimi paragrafi passeremo in rassegna i principali metodi di questa categoria di oggetti, utili per scrivere e leggere i record presenti nelle tabelle del database recuperato.

SQLITE

Il sito di riferimento per SQLite è disponibile all'indirizzo: www.sqlite.org/
La documentazione sulle funzionalità supportate da SQLite e sulla loro sintassi d'uso è all'indirizzo: www.sqlite.org/docs.html

INSERIRE, AGGIORNARE E CANCELLARE I RECORD

Gli oggetti *SQLiteDatabase* espongono un metodo di convenienza per l'inserimento di un nuovo record in una tabella:

```
public long insert(String table, String
                    nullColumnHack, ContentValues values)
```

L'argomento *table* è il nome della tabella in cui inserire il nuovo record; *nullColumnHack* è un valore che nella maggior parte dei casi va posto su *null*, visto che serve solo nel caso particolare in cui si crea un record senza specificare i suoi valori iniziali; *values* è la mappa dei valori iniziali da salvare nel record, di tipo *android.content.ContentValues*. Il metodo restituisce un valore *long*, che è l'ID del record appena creato, oppure è *-1* se il record non è stato creato. Ecco un esempio:

```
SQLiteDatabase db = mioDatabaseHelper.getWritable
                                        Database();
ContentValues values = new ContentValues();

values.put("nome", "Mario");
values.put("cognome", "Rossi");
values.put("telefono", "061299178");
long id = db.insert("agenda", null, values);
```

L'aggiornamento dei dati contenuti in un record è possibile grazie al metodo *update()*, così definito:

```
public int update(String  table, ContentValues
          values, String whereClause, String[] whereArgs)
```

Come nel caso precedente, *table* è il nome della tabella e *values* è la mappa dei campi da aggiornare con i nuovi valori da assegnare. Gli argomenti *whereClause* e *whereArgs* servono per selezionare il record o i record da aggiornare. Sono sostanzialmente delle clausule WHERE di SQL, da comporre secondo le comuni regole del linguaggio.
Si può procedere in due modi. Immaginiamo di volere modificare il numero di telefono in agenda per la voce "*Mario Rossi*".
Il primo modo consiste nello scrivere l'intera clausola di tipo *WHERE* sull'argomento *whereClause*, tenendo su *null* l'argomento *whereArgs*:

```
SQLiteDatabase db = mioDatabaseHelper.getWritable
                                        Database();
ContentValues values = new ContentValues();
values.put("telefono", "068390239");
String whereClause = "nome = 'Mario' AND
                              cognome = 'Rossi'";
int r = db.update("agenda", values, whereClause, null);
```

La seconda tattica consiste invece nell'usare dei segni di punto interrogativo in *whereClause*, a cui far poi corrispondere degli argomenti nell'array di stringhe *whereArgs*:

```
SQLiteDatabase db = mioDatabaseHelper.getWritable
                                        Database();
ContentValues values = new ContentValues();
values.put("telefono", "068390239");

String whereClause = "nome = ? AND cognome = ?";
String[] whereArgs = { "Mario", "Rossi" };
int r = db.update("agenda", values, whereClause,
                                        whereArgs);
```

Questa seconda pratica è generalmente da preferirsi, perché evita le ambiguità (e le vulnerabilità) dovute all'utilizzo degli apici come delimitatori delle stringhe SQL (cercate "sql injection" su Google per approfondire).
È anche possibile passare *null* sia su *whereClause* che su *whereArgs*: in questo caso saranno aggiornati tutti i record presenti nella tabella. In ogni caso, il metodo *update()* restituisce un intero, che indica proprio quanti sono i record aggiornati mediante l'esecuzione dell'istruzione. Per cancellare uno o più record è a disposizione il metodo *delete()*:

```
public int delete(String table, String whereClause,
                                  String[] whereArgs)
```

L'utilizzo di *delete()* è simile ad *update()*, con la sola differenza che in questo caso non ci sono nuovi valori da sovrascrivere ai precedenti, visto che si tratta di un'operazione di cancellazione e non di aggiornamento.
Ad esempio:

```
SQLiteDatabase db = mioDatabaseHelper.
                             getWritable Database();
String whereClause = "_id = ?";
String[] whereArgs = { "1" };
int r = db.delete("agenda", whereClause, whereArgs);
```

Altre istruzioni di scrittura su database, come per esempio quelle di gestione delle tabelle, possono invece essere eseguite passando direttamente per il metodo *execSQL()* descritto al paragrafo precedente.
Ad esempio la tabella "agenda" potrebbe essere cancellata semplicemente chiamando:

```
SQLiteDatabase db = mioDatabaseHelper.getWritable
                                        Database();
db.execSQL("DROP TABLE agenda");
```

TIPI DI DATI IN SQLITE

I tipi di dati supportati da SQLite sono descritti nel documento: www.sqlite.org/datatype3.html

IL METODO QUERY

Le operazioni di lettura e ricerca vanno svolte servendosi di uno dei metodi *query()* messi a disposizione dagli oggetti *SQLiteDatabase*. Il principale di questi è:

```
public Cursor query(String table, String[] columns,
        String selection, String[] selectionArgs, String
        groupBy, String having, String orderBy)
```

Analizziamo insieme i tanti argomenti richiesti:

- *table* è il nome della tabella da interrogare.
- *columns* è la lista con i nomi delle colonne i cui
- valori devono essere inclusi nella risposta. Se *null*
- vengono restituite tutte le colonne della tabella.
- *selection* è la clausola *WHERE*, come nei casi del
- paragrafo precedente. Se *null* vengono restituite
- tutte le righe della tabella.
- selectionArgs è la lista degli argomenti della clau sola *WHERE* al punto precedente.
- *groupBy* è la clausola SQL di raggruppamento. Se
- *null* non si applica alcun raggruppamento.
- *having* è la clausola SQL di restrizione sul raggruppamento. Se *null* non si applicano restrizioni al raggruppamento.
- *orderBy* è la clausola SQL di raggruppamento. Se *null* non si applica un ordinamento specifico alla lista dei risultati restituiti.

Come è possibile osservare, se si mastica un po' di SQL, il metodo *query()* non fa altro che richiedere in argomento le singole parti che costituiscono la classica istruzione SQL di SELECT, del tipo:

```
SELECT … FROM … WHERE … GROUP BY … HAVING …
        ORDER BY …
```

Ad eccezione del nome della tabella da interrogare, tutti gli altri argomenti possono essere nulli. In pratica chiamare:

```
db.query("agenda", null, null, null, null, null, null);
```

è come fare:

```
SELECT * FROM agenda
```

Immaginiamo adesso di voler ricercare gli ID di tutti i "*Rossi*" in agenda, ordinati in maniera crescente. Dovremmo fare:

```
SQLiteDatabase db = mioDatabaseHelper.getReadable
        Database();
String[] columns = { "_id" };
String selection = "cognome = ?";
String[] selectionArgs = { "Rossi" };
String orderBy = "_id ASC";
db.query("agenda", columns, selection, selectionArgs,
        null, null, orderBy);
```

Se preferite esprimere le vostre ricerche in SQL puro, anziché usare il metodo di convenienza *query()*, potete usare direttamente il metodo *rawQuery()*:

```
String query = "SELECT _id FROM agenda WHERE
        cognome = ? ORDER BY _id ASC";
String[] selectionArgs = { "Rossi" };
db.rawQuery(query, selectionArgs);
```

Questa query è equivalente alla precedente, ma espressa completamente in SQL. L'SQL puro torna utile soprattutto nei casi in cui si devono costruire delle query complesse, magari con relazioni tra più tabelle.
Qualunque sia il metodo di interrogazione utilizzato, il risultato sarà sempre un oggetto di classe *android.database.Cursor*. La forma di codice corretta, quindi, è del tipo:

```
Cursor cursor = dq.query(…);
```

oppure:

```
Cursor cursor = db.rawQuery(…);
```

I cursori sono gli oggetti di Android che permettono di navigare all'interno di un *result set*, ossia nell'insieme dei risultati restituiti da una query. Dovete pensare ai risultati di una query come ad una tabella di natura temporanea, organizzata in righe e colonne.
Le colonne sono quelle selezionate dalla vostra istruzione di ricerca, mentre le righe contengono i dati di ciascun risultato individuato. Il cursore è un meccanismo che di volta in volta indica una riga diversa di questa tabella temporanea. Il cursore può dirvi anzitutto quante righe ci sono:

```
int count = cursor.getCount();
```

Appena creato, un cursore non "punta" ad alcuna riga dei risultati. Si può farlo puntare ad una riga usando i suoi metodi di tipo *move*. Il più importante è:

```
public boolean moveToNext()
```

Se si richiama il metodo non appena il cursore è stato creato, si otterrà che il cursore punterà alla prima riga del result set.
Chiamandolo di nuovo, il cursore si sposterà sulla seconda riga, poi sulla terza, sulla quarta e così via. Il metodo restituisce *true* finché ci

SQLITEQUERY BUILDER

La classe *android.database.sqlite.SQLiteQueryBuilder* è una classe di utilità che può essere utilizzata per comporre query di maggiore complessità, in maniera strutturata, senza dover scrivere codice SQL.

sono record disponibili. Nel momento in cui si è sull'ultima riga della tabella dei risultati e si invoca nuovamente *moveToNext()*, il metodo ritornerà *false* per indicare che non ci sono ulteriori risultati su cui muoversi. Attraverso questo metodo è possibile costruire un ciclo di analisi dei risultati di una query basato sul seguente modello:

```
Cursor cursor = db.query(…);
while (cursor.moveToNext()) {
  // analizza la riga corrente
}
```

La classe *Cursor* mette poi a disposizione altri metodi di tipo *move*, che permettono ulteriori tipi di movimento all'interno del result set:

- *public boolean moveToFirst()*
  Porta alla prima riga del result set.

- *public boolean moveToLast()*
  Porta all'ultima riga del result set.

- *public boolean moveToPrevious()*
  Si muove indietro di una riga.

- *public boolean moveToPosition(int position)*
  Si muove ad una riga specifica del result set.

Anche questi quattro metodi restituiscono un booleano, per indicare se l'operazione di spostamento è riuscita oppure no.
Una volta che si è puntata la riga che si intende analizzare, bisogna estrarre da questa i valori presenti per ciascuna colonna.
È possibile farlo con uno dei seguenti metodi:

- *public byte[] getBlob(int columnIndex)*
- *public double getDouble(int columnIndex)*
- *public float getFloat(int columnIndex)*
- *public int getInt(int columnIndex)*
- *public long getLong(int columnIndex)*
- *public short getShort(int columnIndex)*
- *public String getString(int columnIndex)*

I metodi di questo gruppo funzionano tutti alla stessa maniera: restituiscono il valore della colonna con l'indice indicato. Gli indici partono da zero. Significa che se avete richiesto le colonne *"_id"*, *"nome"*, *"cognome"* allora 0 corrisponde a *"_id"*, 1 a *"nome"* e 2 a *"cognome"*. A seconda della natura del dato che si sta estraendo dovrete scegliere il metodo più adatto dall'elenco.
Ad esempio *"nome"* e *"cognome"* vanno recuperati con *getString()*, mentre l'ID è un intero che può essere recuperato con *getInt()* o *getLong()*:

```
SQLiteDatabase db = mioDatabaseHelper.getReadable
                                        Database();
String[] columns = { "_id", "nome", "cognome" };
String orderBy = "_id ASC";
Cursor cursor = db.query("agenda", columns, null, null,
null, null, orderBy);
while (cursor.moveToNext()) {
  long id = cursor.getLong(0);
  String nome = cursor.getString(1);
  String cognome = cursor.getString(2);
}
```

Particolare attenzione deve essere prestata ai campi che potrebbero essere nulli. Si può controllare se il valore in una colonna è nullo adoperando il seguente metodo:

```
public boolean isNull(int columnIndex)
```

Ad esempio:

```
if (cursor.isNull(1)) {
  // campo nullo
} else {
  // campo non nullo
  int c = cursor.getInt(1);
  …
}
```

La classe *Cursor* dispone poi di numerosi altri metodi, che permettono di esplorare più nel dettaglio il result set restituito da una query. La documentazione ufficiale, come sempre, è un ottimo punto di partenza per l'approfondimento.

UN NOTEPAD CON DATABASE

Per approfondire e mettere in pratica l'argomento database, nel CD-Rom allegato alla rivista troverete il codice sorgente di una rivisitazione del blocco note realizzato il mese scorso, che era basato su file di testo. Questa volta il notepad gestisce i propri contenuti utilizzando un database di SQLite.

Carlo Pelliccia

***Fig.1:** L'esempio presente nel CD-Rom è un blocco note che salva i contenuti su database.*

Carlo Pelliccia lavora presso 4IT (www.4it.it), dove si occupa di analisi e sviluppo software per piattaforme Java. Nella sua carriera di technical writer ha pubblicato cinque manuali ed oltre duecento articoli, molti dei quali proprio tra le pagine di ioProgrammo. Il suo sito, che ospita anche diversi progetti Java Open Source, è disponibile all'indirizzo www.sauronsoftware.it

GESTIONE DEI CONTENT PROVIDER

I CONTENT PROVIDER COSTITUISCONO LA MANIERA DI ANDROID PER CONDIVIDERE DATI FRA LE APPLICAZIONI. IN QUESTO ARTICOLO IMPAREREMO A CONSULTARE I PROVIDER PREDEFINITI E VEDREMO ANCHE COME COSTRUIRE UN FORNITORE DI CONTENUTI CUSTOM

Conoscenze richieste
Basi di Java

Software
Java SDK (JDK) 5+, Android SDK, Eclipse 3.3+, ADT

Impegno

Tempo di realizzazione

Nei due numeri precedenti abbiamo imparato ad interagire con il file system ed il DBMS. Come abbiamo visto, secondo i meccanismi di gestione della sicurezza di Android, sia i file che i database sono solitamente di esclusiva proprietà dell'applicazione che li genera. Come fare, allora, per condividere dati strutturati tra più applicazioni Android? La risposta è: mediante i Content Provider.
Un Content Provider è una parte di un'applicazione Android che si occupa di rendere disponibili dei dati alle altre applicazioni installate nel sistema. Ogni applicazione, pertanto, può definire una o più tipologie di dati e rendere poi disponibili tali informazioni esponendo uno o più Content Provider. Nell'ordine inverso, invece, qualunque applicazione può richiedere l'accesso ad un particolare tipo di dato: il sistema la metterà in contatto con il corrispondente *Content Provider* precedentemente installato nel sistema.

RICERCA DEI CONTENUTI

Ogni tipo di contenuto esposto mediante Content Provider viene identificato attraverso un URI, cioè un indirizzo univoco. La forma tipica di questo genere di URI è:

```
content://riferimento-di-base/bla/bla/bla
```

Android dispone di diversi Content Provider built-in, come quello per le immagini o quello per accedere ai contatti in rubrica. L'URI per accedere ai contatti, ad esempio, è:

```
content://com.android.contacts/contacts
```

Siccome questi URI sono un po' arbitrari, per convenienza si è soliti fare in modo che qualche classe riporti l'indirizzo in maniera statica, in modo che non sia necessario digitarlo per esteso. I Content Provider preinstallati in Android sono consultabili al package *android.provider*. In questo pacchetto, ad esempio, si trova la classe *ContactsContract.Contacts*, che è quella che fa da ponte per l'accesso al provider dei contatti in rubrica. Al suo interno c'è la costante statica *CONTENT_URI*, di tipo *android.net.Uri*, che riporta l'URI che identifica univocamente il provider.
Una volta che si conosce l'URI di una tipologia di contenuto, interagire con il provider che la eroga è più o meno come fare delle interrogazioni ad un database. Per prima cosa si deve recuperare un'istanza dell'oggetto *android.content.ContentResolver*. Stando all'interno di una *Activity* (o avendo a disposizione un oggetto *android.app.Context*) si può usare il metodo *getContentResolver()*. Ad esempio:

```
ContentResolver cr = getContentResolver();
```

Gli oggetti *ContentResolver* permettono le interrogazioni attraverso il loro metodo:

```
public Cursor query(Uri uri, String[] projection, String
        selection, String[] selectionArgs, String sortOrder)
```

I parametri da fornire sono i seguenti:

- *uri* è l'indirizzo che identifica il tipo di contenuto ricercato.
- *projection* è la lista con i nomi delle colonne i cui valori devono essere inclusi nella risposta. Se *null*, vengono restituite tutte le colonne disponibili.
- *selection* è la clausola *WHERE*. Se *null*, vengono restituite tutte le righe disponibili.
- *selectionArgs* è la lista degli argomenti della clausola WHERE al punto precedente.
- *sortOrder* è la clausola SQL di ordinamento. Se *null*, non si applica un ordinamento specifico alla lista dei risultati restituiti.

Questo schema, come è possibile notare, ricalca molto da vicino quello conosciuto il mese scorso, quando abbiamo preso in esame i metodi per la ricerca in un database. Anche il tipo del risultato restituito è di tipo a noi noto: si tratta di un oggetto *android.database.Cursor*, che possiamo sfogliare per sondare i record restituiti come risposta alla nostra query. Si faccia

solamente attenzione al fatto che il cursore restituito, in questo caso, potrebbe anche essere nullo: avviene quando l'URI fornito al metodo non corrisponde ad alcun provider registrato nel sistema.
Per comporre query complesse, così come per prendere in esame i risultati restituiti, è necessario conoscere il nome ed il tipo delle colonne che costituiscono lo specifico tipo di dato richiesto. Anche in questo caso si è soliti includere tali informazioni all'interno delle costanti statiche di una classe. Nel caso della rubrica di sistema, ad esempio, le colonne disponibili sono elencate all'interno *ContactsContract.Contacts*. Ecco un esempio di codice che interroga la lista dei contatti di Android, ordinando i risultati in base al loro nome visualizzato:

```
ContentResolver cr = getContentResolver();
Uri uri = Contacts.CONTENT_URI;
String[] projection = { Contacts.DISPLAY_NAME };
String selection = null;
String[] selectionArgs = null;
String sortOrder = Contacts.DISPLAY_NAME + " ASC";
Cursor cursor = cr.query(uri, projection, selection,
                                   selectionArgs, sortOrder);
while (cursor.moveToNext()) {
   String displayName = cursor.getString(0);
   Log.i("Test", displayName);
}
cursor.close();
```

Attenzione a gestire sempre correttamente il ciclo di vita del cursore, senza dimenticarsi di chiuderlo ad utilizzo completato. Se si lavora all'interno di una attività, risulta conveniente sostituire il metodo *query()* di *ContentResolver* con l'analogo *managedQuery()* di *Activity*: in questo caso, infatti, la gestione del cursore viene svolta automaticamente dall'attività.
Per esercizio su quanto appena appreso, realizziamo ora una semplice attività capace di mostrare in una lista il nome di ogni contatto presente in rubrica:

```
package it.ioprogrammo.contentproviderdemo01;

import android.app.ListActivity;
import android.content.Context;
import android.database.Cursor;
import android.net.Uri;
import android.os.Bundle;
import android.provider.ContactsContract.Contacts;
import android.view.View;
import android.view.ViewGroup;
import android.widget.CursorAdapter;
import android.widget.TextView;

public class ContentProviderDemo01Activity extends
                                          ListActivity {

   @Override
   public void onCreate(Bundle savedInstanceState) {
     super.onCreate(savedInstanceState);
     Uri uri = Contacts.CONTENT_URI;
     String[] projection = { Contacts._ID, Contacts.
                                        DISPLAY_NAME };
     String selection = null;
     String[] selectionArgs = null;
     String sortOrder = Contacts.DISPLAY_NAME + "
                                                ASC";
     Cursor cursor = managedQuery(uri, projection,
                      selection, selectionArgs, sortOrder);
     setListAdapter(new CursorAdapter(this, cursor, true)
                                                       {
       @Override
       public View newView(Context context, Cursor
                        cursor, ViewGroup parent) {
         String displayName = cursor.getString(1);
         TextView textView = new TextView(context);
         textView.setText(displayName);
         return textView;
       }
       @Override
       public void bindView(View view, Context context,
                                        Cursor cursor) {
         String displayName = cursor.getString(1);
         TextView textView = (TextView) view;
         textView.setText(displayName);
       }
     });
   }

}
```

Si faccia attenzione al fatto che questa attività, per girare senza incappare in errori, necessita del permesso *android.permission.READ_CONTACTS* (per i dettagli, fate riferimento al box qui accanto).

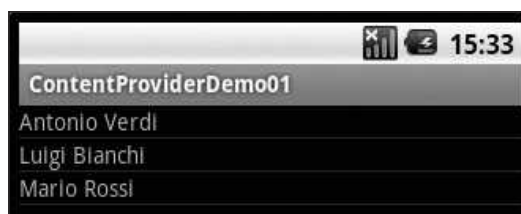



Fig.1: Questa attività si collega al Content Provider della rubrica, riuscendo così a mostrare tutti i contatti registrati nel telefono

NON SOLO RICERCHE

I Content Provider sono in grado di fornire anche funzionalità di inserimento, aggiornamento e cancellazione dei record. La classe *ContentResolver*, oltre al già esaminato metodo *query()*, mette a disposizione i

È PERMESSO?

Il modello di sicurezza di Android fa sì che le applicazioni non possano compiere determinate azioni senza ottenere prima un permesso specifico. Ad esempio, una qualsiasi applicazione non può connettersi alla Rete all'insaputa dell'utente, così come non può manipolare la lista dei contatti in rubrica. Le applicazioni che vogliono compiere questo genere di attività devono dichiararlo esplicitamente. L'utente, quando installa l'applicazione, viene così informato di quali sono le operazioni potenzialmente pericolose che il software può eseguire, ed è così libero di accordare o meno la propria fiducia al produttore dell'app.
Dal punto di vista dello sviluppo, un permesso può essere richiesto aggiungendo nell'*AndroidManifest.xml* dell'applicazione un tag del tipo: *<uses-permissionandroid:name="permesso_richiesto" />*
L'elenco dei permessi a disposizione è flessibile ed espandibile. La documentazione ufficiale riporta i permessi built-in in Android. Usando un editor avanzato, come Eclipse, è poi possibile ottenere una lista visuale di tutti i permessi disponibili nel sistema per cui si sta sviluppando.

RICERCA PER ID

Tutti i contenuti esposti mediante provider dovrebbero avere una colonna chiamata *_ID*, con l'identificativo numerico del record, univoco nella sua categoria. Se si conosce l'ID di un record e lo si vuole estrarre direttamente dal suo provider, si può fare ricorso alla classe *android.content.ContentUris* e al suo metodo statico *withAppendedId()*: *Uri uri = ContentUris.withAppendedId(uri_di_base_del_contenuto, id_del_contenuto); Cursor c = cr.query(uri, null, null, null, null);*

seguenti altri metodi:

- **public Uri insert(Uri uri, ContentValues values)**
  Inserisce un nuovo record del tipo specificato mediante il parametro *uri*. I valori per i campi del nuovo record devono essere specificati attraverso la mappa *values*, di tipo *android.content.ContentValues*. Il metodo restituisce l'URI di dettaglio assegnato all'elemento appena inserito.

- **public int update(Uri uri, ContentValues values, String where, String[] selectionArgs)**
  Aggiorna uno o più record del tipo specificato mediante il parametro *uri*. La selezione avviene attraverso l'uso combinato dei parametri *where* e *selectionArgs*. I nuovi valori da assegnare ai record selezionati devono essere specificati attraverso la mappa values, di tipo *android.content.ContentValues*. Il metodo restituisce il numero dei record aggiornati.

- **public int delete(Uri uri, String where, String[] selectionArgs)**
  Cancella uno o più record del tipo specificato mediante il parametro *uri*. La selezione avviene attraverso l'uso combinato dei parametri *where* e *selectionArgs*. Il metodo restituisce il numero dei record cancellati.

Implementiamo un esempio pratico. Questa volta lavoreremo con la lista delle immagini memorizzate nella galleria del telefono. L'URI di base per l'accesso alle immagini che sono nello storage esterno viene riportato nella proprietà:

```
android.provider.MediaStore.Images.Media.EXTERNAL_
                                        CONTENT_URI
```

Tutte le immagini recuperate mediante l'apposito provider dispongono di una colonna *"_ID"*, in cui viene riportato il loro identificativo numerico univoco. Come indicato nel box laterale, dato l'ID di un contenuto, è possibile avere un suo URI specifico facendo:

```
Uri uri = ContentUris.withAppendedId(uri_generico, id);
```

Dato l'URI puntuale di un contenuto-immagine, è possibile caricare l'immagine (sotto forma di oggetto *android.graphics.Bitmap*) facendo:

```
ContentResolver cr = new ContentResolver(this);
Bitmap image = MediaStore.Images.Media.getBitmap(cr,
                                              uri);
```

Sfruttando queste conoscenze, andiamo a realizzare un'attività in grado di svolgere i seguenti compiti:

1. Interrogare il Content Provider delle immagini su storage esterno, per ottenerne l'elenco.
2. Mostrare le immagini, facendo uso di un widget *android.widget.Gallery* (cfr. ioProgrammo 151).
3. Al clic su una delle immagini, eseguire la cancellazione della medesima, previa conferma da parte dell'utente.

Tradotto in codice:

```
package it.ioprogrammo.contentproviderdemo02;
...
public class ContentProviderDemo02Activity extends
Activity {
  private static final int DELETE_DIALOG = 1;
  private Gallery gallery = null;
  private int selectedImageId;

  @Override
  public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    gallery = new Gallery(this);
    Uri uri = MediaStore.Images.Media.EXTERNAL_
                                        CONTENT_URI;
  ...
```

CREARE UN CONTENT PROVIDER

Se volete condividere i dati della vostra applicazione con gli altri software installati nel telefono, potete implementare il vostro Content Provider. Farlo è molto semplice: bisogna estendere la classe *android.content.ContentProvider*, che richiede l'implementazione dei seguenti metodi:

- **public boolean onCreate()**
  Il codice da eseguirsi alla creazione del provider. Il metodo deve restituire un booleano: *true*, per segnalare che la creazione del provider è andata a buon fine; *false*, in caso contrario.
- **public String getType(Uri uri)**
  Dato un URI di gestione del provider, questo metodo deve restituire il tipo MIME del contenuto corrispondente. Solitamente, con tipi di dati personalizzati, non bisogna far altro che inventare il nome della tipologia, seguendo però alcuni criteri. Se l'URI specificato corrisponde ad un contenuto specifico (in genere avviene quando l'URI contiene l'ID del contenuto), allora bisogna restituire un tipo MIME del tipo:

```
vnd.android.cursor.item/vnd.il_nome_del_tipo
```

Per gli URI che corrispondono a gruppi di contenuti (senza ID, quindi), la formula è del tipo:

```
vnd.android.cursor.dir/vnd.il_nome_del_tipo
```

- **public Cursor query(Uri uri, String[] projection, String selection, String[] selectionArgs, String sortOrder)**
  Il metodo richiamato per eseguire una ricerca tra i dati gestiti dal provider.
- **public insert(Uri uri, ContentValues values)**
  Il metodo richiamato per eseguire un inserimento nei dati gestiti dal provider.
- **public int update(Uri uri, ContentValues values, String selection, String[] selectionArgs)**
  Il metodo richiamato per eseguire un aggiornamento dei dati gestiti dal provider.
- **public int delete(Uri uri, String selection, String[] selectionArgs)**
  Il metodo richiamato per eseguire una cancellazione fra i dati gestiti dal provider.

Una volta che il Content Provider è stato implementato, bisogna registrarlo nel file *AndroidManifest.xml*, osservando il seguente modello:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest ...>
  <application ...>
    <provider android:name="MioContentProvider"
      android:authorities="mio_dominio/mio_tipo_di_
                                               dato" />
  </application>
</manifest>
```

L'attributo *name* serve per specificare il nome della classe che implementa il provider; l'attributo

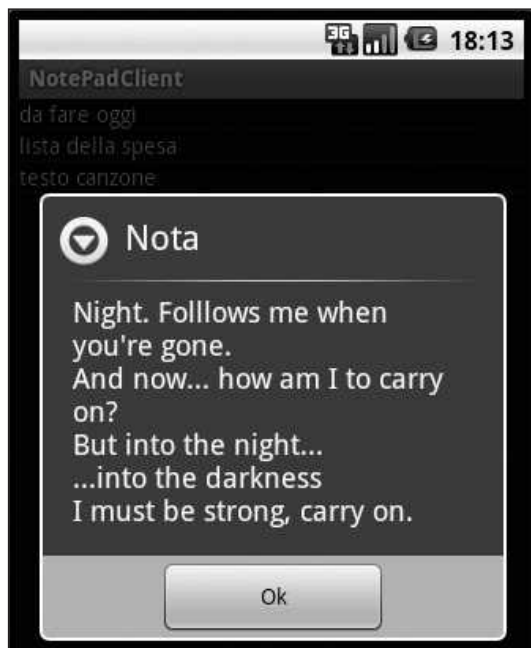



Fig.2: Le note generate dall'applicazione del mese scorso vengono ora consultate attraverso una seconda applicazione, passando per un Content Provider

authorities, invece, definisce la parte fondamentale dell'URI gestito dal provider, cioè quello che dovranno utilizzare le altre applicazioni per interagire con il nostro Content Provider. Ad esempio, facciamo il caso che *autorities* sia:

```
it.ioprogrammo.arretrati
```

In questo caso, il Content Provider riceverà tutte le richieste il cui URI sia del tipo:

```
content:// it.ioprogrammo.arretrati/bla/bla/bla
```

Per concludere, è poi necessario comunicare a chi dovrà utilizzare il provider quale sia il suo URI di base e quali i nomi ed i tipi delle colonne di ciascun tipo di contenuto gestito. Solitamente conviene realizzare una o più classi che contengano queste informazioni, da rendere poi disponibili a chi dovrà servirsi del provider.
Proviamo un esempio. Recuperiamo il progetto del blocco note realizzato nel numero precedente e andiamo ad arricchirlo con un Content Provider, capace di esportare verso l'esterno le note gestite dall'applicazione:

```
package it.ioprogrammo.notepad;
...
public class NotePadContentProvider extends
                                        ContentProvider {
...
```

Registriamo il provider nel file *AndroidManifest.xml* dell'applicazione:

```
<provider android:name="NotePadContentProvider"
        android:authorities="it.ioprogrammo.notepad" />
```

Aggiornate ora l'applicazione sul vostro smartphone o nell'emulatore. Da questo momento in poi l'elenco delle note è gestibile non solo attraverso l'attività incorporata dall'applicazione stessa, ma anche attraverso il Content Provider che risponde a partire dall'URI:

```
content://it.ioprogrammo.notepad
```

Per provare, implementate una nuova applicazione Android, estranea alla precedente, al cui interno va inserita la seguente attività:

```
package it.ioprogrammo.notepadclient;
...
public class NotePadClientActivity extends ListActivity
  {
  ...
```

Carlo Pelliccia

LA CLASSE URIMATCHER

Uno strumento molto utile quando si costruiscono dei Content Provider complessi è la classe *android.content.UriMatcher*. Questa utilità permette di lavorare più agilmente con gli oggetti Uri di Android, discriminando facilmente l'area di appartenenza di ciascun indirizzo ricevuto da un provider di contenuti.

Carlo Pelliccia lavora presso 4IT (www.4it.it), dove si occupa di analisi e sviluppo software per piattaforme Java. Nella sua carriera di technical writer ha pubblicato cinque manuali ed oltre duecento articoli, molti dei quali proprio tra le pagine di ioProgrammo. Il suo sito, che ospita anche diversi progetti Java Open Source, è disponibile all'indirizzo www.sauronsoftware.it

LE APPLICAZIONI GIRANO IN PARALLELO

I SERVIZI SONO QUELLA FUNZIONALITÀ DI ANDROID CHE PERMETTE DI ESEGUIRE OPERAZIONI IN SOTTOFONDO, ANCHE QUANDO L'APPLICAZIONE CHE LE HA AVVIATE NON È PIÙ ATTIVA. INSOMMA: MULTITASKING ALLO STATO PURO, ANCHE IN MOBILITÀ!

Nella prima puntata di questo corso (apparsa su ioProgrammo 143) si è subito spiegato come le applicazioni Android contengano sempre uno o più componenti scelti fra i quattro tipi fondamentali: le attività, i servizi, i broadcast receiver e i content provider. Delle attività ci siamo occupati a lungo, e continueremo a farlo in futuro. Nel numero precedente, invece, sono stati introdotti i content provider, oggetti indispensabili per la condivisione dei dati fra più applicazioni. Oggi ci occuperemo dei servizi.

COS'È UN SERVIZIO

Un servizio, nel gergo di Android, è una parte di una applicazione che gira in background. A differenza delle attività, i servizi non dispongono di un'interfaccia grafica attraverso la quale interagire con l'utente. Allo stesso tempo, però, i servizi non vanno confusi né con i processi né con i thread, che sono cose distinte e ben diverse. Un servizio, infatti, è gestito direttamente da Android, e per questo possiede un proprio peculiare ciclo di vita. In più i servizi si avvantaggiano delle interfacce di programmazione messe a disposizione da Android, attraverso le quali possono integrarsi profondamente con il sistema sottostante.

Un servizio è necessario ogni volta che un'applicazione deve fare qualcosa in background, senza occupare lo schermo. Si pensi, ad esempio, ad un'applicazione tipo un multimedia player, in grado di riprodurre album musicali in formato MP3. Un software di questo tipo disporrà sicuramente di una o più attività per interagire con l'utente, in modo che sia possibile selezionare un album musicale ed avviarne la riproduzione. Una volta che l'esecuzione ha avuto inizio, però, non è buona cosa che l'attività del player rimanga sempre e costantemente sul display dello smartphone. L'utente, ad esempio, potrebbe voler navigare in Internet, giocare o fare altro ancora, senza però interrompere l'ascolto. Ecco allora che è conveniente fare in modo che l'esecuzione della playlist avvenga in sottofondo, in modo che l'attività di interfaccia del player multimediale possa essere rimossa dallo schermo. Lo si può fare, naturalmente, con un servizio.

Ecco un altro esempio, per chiarire ancora meglio il concetto: immaginiamo di dover realizzare un'applicazione che, di tanto in tanto, si colleghi ad Internet per scaricare via Web delle notizie. L'utente ha imposto un filtro sulle notizie di suo interesse, specificando delle parole chiave: solo le news che contengono le keyword specificate gli devono essere notificate. È possibile farlo avviando un servizio che, in sottofondo, si colleghi alla rete e analizzi le notizie di volta in volta disponibili. Quando si incontra una notizia in grado di soddisfare il filtro impostato, il servizio non deve far altro che lanciare un'attività per notificare l'evento all'utente. Con i servizi di Android è possibile fare anche questo.

CREARE UN SERVIZIO

La prima cosa da farsi per realizzare un servizio è estendere la classe *android.app.Service*:

```
import android.app.Service;
public class MioServizio extends Service {
  // ...
}
```

La classe *Service* richiede l'implementazione del metodo astratto *onBind()*. Si tratta di un metodo necessario quando il servizio che si sta realizzando dovrà essere reso pubblico. Altre applicazioni installate nel sistema, in questo caso, potranno collegarsi con il servizio (operazione definita, per l'appunto, *bind*) e interagire con esso. Android comprende alcuni servizi che offrono questa caratteristica, come ad esempio quello utile per acquisire le coordinate geografiche dal ricevitore GPS.

Se non interessa sviluppare un servizio in grado di interagire in maniera avanzata con le applicazioni esterne, come nei casi più semplici, la cosa migliore da fare è implementare *onBind()* alla seguente maniera:

```
import android.content.Intent;
```

```
import android.os.IBinder;
// ...
@Override
public IBinder onBind(Intent intent) {
  return null;
}
```

Il passo successivo consiste nel registrare il nuovo servizio nel file *AndroidManifest.xml*:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest ...>
  <application ...>
    <service android:name="MioServizio" />
  </application>
</manifest>
```

Lo scheletro del servizio, a questo punto, è già pronto.

AVVIARE ED ARRESTARE UN SERVIZIO

I servizi, per entrare in funzione, devono essere avviati. È possibile avviare un servizio dall'interno di un'attività, di un content provider o di un altro servizio. La classe *android.content.Context*, da cui derivano tutti i mattoni fondamentali della programmazione Android, mette a disposizione i seguenti metodi per il controllo dei servizi:

- **public ComponentName startService(Intent service)**
  Avvia il servizio specificato attraverso il parametro di tipo *android.content.Intent*. Restituisce un oggetto *android.content.ComponentName* che riporta il nome di dettaglio del servizio avviato.
- **public boolean stopService(Intent service)**
  Arresta il servizio specificato attraverso il parametro di tipo *android.content.Intent*. Restituisce un booleano per indicare se l'operazione è riuscita (*true*) o meno (*false*).

Immaginiamo di aver implementato il servizio contenuto nella classe *MioServizio*, registrato poi correttamente nell'*AndroidManifest.xml* dell'applicazione, come indicato nel paragrafo precedente. A questo punto, da un qualsiasi altro componente della medesima applicazione, come ad esempio un'attività, è possibile avviare il servizio chiamando semplicemente:

tartService(new Intent(this, MioServizio.class));

Il servizio può poi essere arrestato chiamando analogamente:

stopService(new Intent(this, MioServizio.class));

CICLO DI VITA DI UN SERVIZIO

Ora che siamo in grado di creare l'ossatura di un servizio e di gestire l'avvio e l'arresto dello stesso, andiamo a imparare come inserire al suo interno la nostra logica di programmazione, che il servizio dovrà poi eseguire in sottofondo. La prima cosa da comprendere è il ciclo di vita dei servizi Android, che tra l'altro è molto semplice e lineare, perlomeno fino a quando il servizio è usato a livello elementare. Quando il servizio viene creato ed avviato, il sistema chiama automaticamente il suo metodo:

public void onCreate()

La chiusura di un servizio, invece, viene gestita mediante il metodo:

public void onDestroy()

Un servizio può venire arrestato e distrutto in tre casi:

1. Quando, come abbiamo visto in precedenza, un altro componente della stessa applicazione chiama *stopService()*.
2. Un servizio può arrestarsi da solo, chiamando il suo stesso metodo *stopSelf()*.
3. Come nel caso delle attività, un servizio può essere arrestato dal sistema nel momento in cui le risorse scarseggiano ed è necessario liberare il processore e la memoria.

Naturalmente i metodi *onCreate()* e *onDestroy()* possono essere ridefiniti a nostro piacimento, per fare in modo di intercettare gli eventi e mettere quindi in moto la nostra logica.

UNA DIMOSTRAZIONE CONCRETA

Facciamo una prova con un servizio così definito:

```
package it.ioprogrammo.servicedemo01;

import java.util.Timer;
import java.util.TimerTask;
...
```

Questo servizio, all'avvio, usa le classi *java.util.Timer* e *java.util.TimerTask* per compiere ciclicamente un'operazione. Ogni cinque secondi viene immessa la scritta "Servizio in esecuzione" nel log di Android. L'operazione si ripete fino all'arresto del servizio. In più il servizio annota sul log anche i propri cambi di stato (*create* e *destroy*). Serviamoci ora di una semplice attività per disporre dei comandi utili per avviare ed interrompere il servizio:

ONLOWMEMORY()

Se il telefono è a corto di memoria, il vostro servizio riceverà una chiamata sul metodo *onLowMemory()*. Potete ridefinire questo metodo per intercettare l'evento e cercare così di collaborare con il sistema che ospita la vostra applicazione. Se il servizio, ad esempio, tiene in memoria un grosso quantitativo di dati, qui potreste rilasciarne alcuni, per fare spazio alle altre applicazioni che ne hanno bisogno.

LANCIARE UN'ATTIVITÀ DA UN SERVIZIO

I servizi, come si è visto, non hanno interfaccia grafica e non possono pertanto interagire direttamente con l'utente. Può però capitare che un servizio che fino ad un certo momento ha girato in background abbia improvvisamente bisogno di chiedere o mostrare qualcosa all'utente. Per farlo si può lanciare un'attività. Tutti i servizi possono farlo, chiamando il metodo *startActivity()*.

```
package it.ioprogrammo.servicedemo01;
...
public class ServiceDemo01Activity extends Activity {
  @Override
  public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    Button bottoneAvvia = new Button(this);
    bottoneAvvia.setText("Avvia il servizio");
    bottoneAvvia.setOnClickListener(new View.
                                    OnClickListener() {
      @Override
      public void onClick(View v) {
        avviaServizio();
      }
    });
...
```

L'attività è molto semplice: non fa altro che disporre sul display due bottoni utili, rispettivamente, per avviare ed arrestare il servizio realizzato al passo precedente. Anche l'attività, inoltre, logga i propri cambi di stato (*create* e *destroy*). Non resta che assemblare l'applicazione attraverso il file *AndroidManifest.xml*:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/
                                    apk/res/android"
  package="it.ioprogrammo.servicedemo01"
                              android:versionCode="1"
  android:versionName="1.0">
  <application android:icon="@drawable/icon"
                      android:label="@string/app_name">
    <activity android:name=".ServiceDemo01Activity"
      android:label="@string/app_name">
      <intent-filter>
        <action android:name="android.intent.action.
                                              MAIN" />
        <category android:name="android.intent.
                                category.LAUNCHER" />
      </intent-filter>
    </activity>
    <service android:name="MioServizio" />
  </application>
</manifest>
```

Lanciate l'applicazione e monitoratene il log. Avviate ed arrestate il servizio quante volte volete.

Fig.1: La semplice GUI realizzata per avviare ed arrestare il servizio di prova

Provate inoltre ad avviare il servizio e ad abbandonare l'attività per fare altro. Vi capiterà di notare che l'attività può essere distrutta, ma il servizio resterà lo stesso in esecuzione in background.

WALLPAPER CHANGER

Grazie all'esempio appena esaminato siamo ora in grado di programmare un reale servizio Android, secondo le nostre esigenze e la nostra fantasia. Sfruttiamo allora le conoscenze acquisite per realizzare un'applicazione un po' meno didattica ed un po' più concreta: un wallpaper changer automatico, cioè un'applicazione che ogni tanto (ad esempio ogni minuto) cambi automaticamente l'immagine di sfondo del telefono.Per cominciare, create nel vostro ambiente di sviluppo il progetto "*WallpaperChanger*". Il package di riferimento su cui lavoreremo è *it.ioprogrammo.wallpaperchanger*.

Fig.2: Come dimostra questo log, il ciclo di vita del servizio e quello dell'attività che abbiamo avviato sono separati: l'attività può essere distrutta, ma il servizio resta lo stesso in esecuzione

Le immagini che, a rotazione, verranno impostate come wallpaper, saranno parte del software stesso. Preparate quindi una serie di immagini JPEG di dimensione idonea, ed aggiungetele poi al percorso di progetto *assets/wallpapers*. Potete dare alle immagini il nome che preferite. Il cuore dell'applicazione, naturalmente, è un servizio:

```
package it.ioprogrammo.wallpaperchanger;
...
public class WallpaperChangerService extends Service {
  public static boolean STARTED = false;
  private String[] availableWallpapers;
  private int currentWallpaperIndex;
  private Timer timer;
...
```

L'ossatura è la stessa dell'esempio precedente, con un *Timer* ed un *TimerTask* impiegati per compiere ciclicamente l'operazione di aggiornamento del wallpaper. Le immagini disponibili vengono lette dal percorso assets/wallpaper servendosi di un *android.content.res.AssetManager*, che viene recuperato con il metodo *getAssets()*. Ogni 60 secondi viene selezionata l'imma-

gine successiva dell'elenco letto inizialmente. L'oggetto *android.graphics.Bitmap* corrispondente al wallpaper da mostrare, viene caricato servendosi della classe di utilità *android.graphics.BitmapFactory*. Impostare un oggetto *Bitmap* come wallpaper è davvero molto semplice in Android: basta servirsi del singleton *android.app.WallpaperManager* e del suo metodo *setBitmap()*. Infine il servizio memorizza il proprio stato nella proprietà statica *STARTED*, che ci tornerà utile a breve. Ora che il servizio è pronto ci serve un'attività per comandarne l'avvio e l'arresto. Facciamo le cose per bene e serviamoci di file XML per la definizione delle stringhe e dei layout. Cominciamo con il file *values/strings.xml*:

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <string name="app_name">Sfondi automatici</string>
    <string name="startService">Gestisci gli sfondi</string>
    <string name="stopService">Smettila di gestire gli
                                        sfondi</string>
    <string name="finish">Nascondi</string>
</resources>
```

Queste stringhe sono utilizzate nel layout da definire al percorso *layout/main.xml*:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.
                                com/apk/res/android"
    android:orientation="vertical"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent">
    ...
```

In questo layout viene definita una semplicissima interfaccia fatta di tre bottoni: il primo per avviare il servizio di cambio automatico del wallpaper, il secondo per arrestarlo ed il terzo per nascondere l'attività che comprende l'interfaccia stessa. Andiamo a realizzare tale attività:

WallpaperChanger
src
gen [Generated Java Files]
Android 2.1-update1
assets
wallpapers
01.jpg
02.jpg
03.jpg
04.jpg

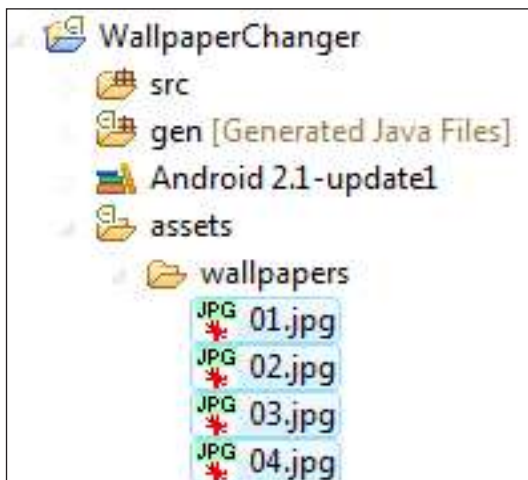



Fig.3: *I wallpaper usati a rotazione dell'applicazione vanno inseriti al percorso di progetto assets/wallpapers*

```
package it.ioprogrammo.wallpaperchanger;
...
public class WallpaperChangerActivity extends Activity
{
    private Button bStartService;
    private Button bStopService;
    private Button bFinish;
    @Override
    public void onCreate(Bundle savedInstanceState) {
...
```

Anche in questo caso il modello utilizzato viene dall'esempio del paragrafo precedente. Rispetto al caso precedente, qui in più si consulta lo stato del servizio, in modo da abilitare e disabilitare secondo necessità i primi due pulsanti dell'interfaccia. Inoltre questa attività può concludersi da sola grazie al terzo bottone inserito, il cui listener associato richiama il metodo *finish()* dell'attività. Per il resto, niente di nuovo.

Fig.4: *L'attività consulta lo stato del servizio per abilitare o disabilitare i bottoni di avvio ed arresto*

Non resta che mettere insieme il tutto nel file *AndroidManifest.xml*:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/
                                    apk/res/android"
    package="it.ioprogrammo.wallpaperchanger"
                              android:versionCode="1"
    android:versionName="1.0">
...
```

Siccome le applicazioni Android non possono cambiare il wallpaper del telefono senza dichiararlo esplicitamente, in accordo con il modello di sicurezza del sistema, nel manifest dell'applicazione è stato necessario dichiarare l'uso del permesso *android.permission.SET_WALLPAPER*.
Ora è tutto pronto, non resta che installare l'applicazione su un emulatore o su uno smartphone ed avviare il servizio di cambio automatico del wallpaper.

Carlo Pelliccia

STARTFOREGROUND

Chiamando il metodo *startForeground()* si può indicare che il servizio è importante e che il sistema dovrebbe cercare di non distruggerlo mai, anche in caso di necessità di memoria. Ciò non significa che il servizio non sarà mai distrutto automaticamente, ma soltanto che la sua priorità sarà innalzata e che si cercherà di evitare l'evento nei limiti del possibile. Nel caso in cui un servizio così impostato dovesse comunque essere distrutto dal sistema, all'utente verrà mostrata una notifica informativa, il cui ID ed i cui contenuti vanno forniti come parametri del metodo *startForeground()*. In questa maniera, ad esempio, l'utente può essere informato del perché il suo MP3 ha smesso di essere riprodotto d'improvviso!

Carlo Pelliccia lavora presso 4IT (www.4it.it), dove si occupa di analisi e sviluppo software per piattaforme Java. Nella sua carriera di technical writer ha pubblicato cinque manuali ed oltre duecento articoli, molti dei quali proprio tra le pagine di ioProgrammo. Il suo sito, che ospita anche diversi progetti Java Open Source, è disponibile all'indirizzo www.sauronsoftware.it

TU SEI QUI! TE LO DICE ANDROID

I SERVIZI LOCATION-BASED SONO UNA DELLE CARATTERISTICHE PIÙ ATTRAENTI DI ANDROID. IMPARIAMO A REALIZZARE APPLICAZIONI IN GRADO DI LOCALIZZARE L'UTENTE VIA GPS E DI DISEGNARE LA SUA POSIZIONE IN UNA MAPPA

Un'applicazione o un servizio possono definirsi location-based quando lavorano con il posizionamento geografico dell'utente. Google è stato un pioniere dei servizi location-based: Google Maps e Google Earth hanno spianato la strada ad una nuova generazione di applicazioni basate sulla geografia e sul posizionamento. Con Android, Google non smette di innovare. È con l'avvento dei dispositivi mobili, infatti, che le applicazioni location-based possono finalmente sfruttare a pieno tutte le loro potenzialità. In questo articolo scopriremo come è facile mostrare mappe, cercare luoghi e interagire con il ricevitore GPS del telefono.

LOCATION MANAGER E PROVIDER

Gli smartphone implementano solitamente uno o più meccanismi di localizzazione dell'utente. Il GPS è quello più noto ed utilizzato, ma non è l'unico meccanismo disponibile. I dispositivi di ultima generazione, infatti, sono in grado di localizzare la propria posizione verificando le reti GSM e Wi-Fi disponibili nei paraggi (in questo caso si parla di localizzazione *network-based*). Si tratta di un meccanismo di localizzazione meno accurato rispetto al GPS, ma comunque affidabile quando è sufficiente conoscere l'area dove si trova l'utente, e non si ha bisogno di calcolare la sua posizione precisa. Android offre un'interfaccia di programmazione che si astrae dal meccanismo di localizzazione utilizzato in dal device. Tale interfaccia è offerta sotto forma di servizio di sistema. Dall'interno di un'attività, i servizi di sistema possono essere recuperati usando il metodo *getSystemService()*. Il metodo richiede in argomento l'identificativo del servizio di sistema desiderato. Gli ID dei servizi di sistema sono conservati in alcune costanti. Quello per il servizio di localizzazione è riportato nella costante *LOCATION_SERVICE*. Il metodo *getSystemService()*, una volta recuperato il servizio, lo restituisce sotto forma di semplice *Object*. È dunque necessario eseguire un casting verso la classe reale del servizio, che in questo caso è *android.location.LocationManager*. Riassumendo, dall'interno di un'attività, il servizio di localizzazione si recupera alla seguente maniera:

```
LocationManager locationManager = (LocationManager)
                    getSystemService(LOCATION_SERVICE);
```

I meccanismi di localizzazione disponibili sono classificati come *provider*. I due provider discussi in precedenza sono identificati con due costanti:

- *LocationManager.GPS_PROVIDER*, il provider collegato al ricevitore GPS del telefono.
- *LocationManager.NETWORK_PROVIDER*, il provider che localizza in maniera approssimativa il telefono basandosi sulle reti raggiungibili.

Prima di usare uno di questi due provider, bisogna verificare che sia supportato dal telefono che esegue l'applicazione. Per farlo, si può richiamare il metodo *getProvider()* di *LocationManager*. Il metodo restituisce *null* se il provider non è disponibile, mentre restituisce un risultato di tipo *android.location.LocationProvider* nel caso lo sia. Ad esempio:

```
LocationProvider gpsProvider = locationManager.
            getProvider(LocationManager.GPS_PROVIDER);
if (gpsProvider == null) {
  // GPS non disponibile
} else { // GPS disponibile }
```

Gli oggetti *LocationProvider*, quando disponibili, forniscono informazioni sul provider richiesto, come la sua accuratezza e le caratteristiche supportate. Dopo essersi accertati che un provider è disponibile, altrettanto importante è controllare che sia anche attivo. L'utente, infatti, potrebbe averlo disabilitato dalle impostazioni del telefono. L'abilitazione di un provider può essere controllata con il metodo *isProviderEnabled()* di *LocationManager*. Ad esempio:

```
boolean gpsEnabled = locationManager.isProvider
                Enabled(LocationManager.GPS_PROVIDER);
```

Prima di usare uno dei due provider, quindi, è meglio controllarne lo stato, in modo da suggerire all'utente di abilitarlo prima di procedere. Ancora un'importante avvertimento: per utilizzare correttamente i servizi di localizzazione, è necessario che l'applicazione, nel proprio *manifest*, richieda l'uso dei permessi *android.permission.ACCESS_COARSE_LOCATION* (per la localizzazione basata sui network) e/o *android.permission.ACCESS_FINE_LOCATION* (per la localizzazione GPS):

```
<?xml version="1.0" encoding="utf-8"?>
<manifest ...>
  <application ...>
    ...
```

DOVE MI TROVO?

Passiamo a vedere come sia possibile recuperare la posizione fornita da uno dei provider previsti. Le misurazioni vengono rappresentate mediante oggetti di tipo *android.location.Location*. I principali e più importanti metodi messi a disposizione da questa classe sono:

- **public long getTime()**
  Restituisce la data in cui l'informazione è stata calcolata, come UNIX timestamp (il numero di millisecondi trascorsi dal 1 Gennaio 1970 alle ore 00:00:00 GMT)
- **public double getLatitude()**
  Restituisce la misura di latitudine.
- **public double getLongitude()**
  Restituisce la misura di longitudine.
- **public boolean hasAltitude()**
  Restituisce *true* se la misurazione comprende anche un dato di altitudine.
- **public double getAltitude()**
  Se disponibile, restituisce la misura di altitudine.
- **public boolean hasSpeed()**
  Restituisce true se la misurazione comprende anche un dato di velocità.
- **public float getSpeed()**
  Se disponibile, restituisce la misura di velocità. L'unità di misura è m/s (metri al secondo).

Una maniera semplice e diretta per ottenere una misurazione è domandarla al *LocationManager*, servendosi del metodo *getLastKnownLocation()*. Il metodo richiede l'ID del provider da interrogare e restituisce un oggetto *Location*. Ad esempio:

```
Location location = locationManager.getLastKnown
            Location(LocationManager.GPS_PROVIDER);
```

Si faccia attenzione al fatto che *getLastKnownLocation()* non calcola la posizione del telefono nel momento in cui lo si richiama: il metodo si limita a restituire l'ultima misurazione disponibile per il provider richiesto. Può accadere che un provider sia stato disabilitato in un posto e riabilitato poi a chilometri di distanza, e che quindi la sua ultima misurazione sia completamente inservibile. Per avere "dati freschi" da uno dei provider, gli si deve esplicitamente chiedere di inviarli non appena disponibili. Il metodo utile per farlo è:

public void requestLocationUpdates(String provider, long minTime, float minDistance, LocationListener listener)

Gli argomenti hanno il seguente scopo:

- **provider** è l'ID del provider (ad esempio *LocationManager.GPS_PROVIDER* o *LocationManager.NETWORK_PROVIDER*).
- **minTime** è il numero di secondi minimo che deve intercorrere tra una misurazione e la successiva. Con questo parametro si può evitare che dal provider vengano troppe informazioni, chiedendo di non mandare una nuova lettura se non è passato almeno un certo numero di secondi.
- **minDistance** è la distanza minima, in metri, che deve esserci tra una misurazione e la successiva. Anche in questo caso l'argomento serve per filtrare le letture del provider ed evitare che da questo vengano troppe informazioni, chiedendo di non mandare una nuova lettura se non si è riscontrato uno spostamento di un certo numero di metri.
- **listener** è l'oggetto, di tipo *android.location.LocationListener*, su cui verranno notificate tutte le misurazioni effettuate e filtrate secondo i parametri precedenti.

Ad esempio:

```
locationManager.requestLocationUpdates(Location
  Manager.GPS_PROVIDER, 5, 1, myLocationListener);
```

Con questa richiesta si sottoscrive il provider GPS e si ricevono le letture e gli eventi provenienti da questo. Le letture, in particolar modo, saranno filtrate: solo se lo spostamento equivale ad almeno un metro, e comunque non più di una notifica ogni cinque secondi. Letture ed eventi saranno segnalati sull'oggetto *myLocationListener*, che naturalmente deve implementare l'interfaccia *LocationListener*. L'implementazione di *LocationListener* richiede la definizione dei seguenti metodi:

- **public void onStatusChanged(String provider, int status, Bundle extras)**
  Notifica un cambio di stato nel provider in argomento. Il nuovo stato (argomento status) può essere: *LocationProvider.OUT_OF_SERVICE*, se il provider è andato fuori servizio; *LocationProvider.TEMPORARILY_UNAVAILABLE*, se il provider è diventato temporaneamente non disponibile; *LocationProvider.AVAILABLE*, se il provider è tornato

GLOBAL POSITIONING SYSTEM

GPS è una di quelle sigle che tutti conoscono ma pochi comprendono. GPS sta per *Global Positioning System*, ossia sistema di posizionamento globale. Si tratta di un sistema costituito da una serie di satelliti artificiali in orbita intorno al pianeta. Ciascun satellite trasmette ciclicamente dei messaggi verso la superficie. I messaggi contengono il segnale orario e delle informazioni sulle orbite percorse. Il ricevitore GPS ascolta questi messaggi e li elabora. In base al ritardo dei segnali e ai contenuti dei messaggi, il ricevitore è in grado di calcolare la propria distanza da ciascun satellite. Nel momento in cui il ricevitore riesce ad agganciare il segnale di quattro o più satelliti, diventa possibile applicare un calcolo matematico simile, nel principio, alla triangolazione. In questa maniera, il ricevitore può determinare la propria posizione sul globo terracqueo, esprimendola in termini di *latitudine* e *longitudine*. Più sono i satelliti di cui il dispositivo riceve il segnale, più è accurata la posizione calcolata.

***Fig.1:** Dalle impostazioni di sistema è possibile abilitare o disabilitare i servizi di localizzazione integrati. Per questo bisogna sempre verificare che siano attivi, prima di utilizzarli*

COORDINATE GPS DA RIGA DI COMANDO

Se non utilizzate Eclipse, potete fornire false coordinate GPS all'emulatore usando la riga di comando. Aprite il prompt dei comandi e connettetevi all'emulatore via telnet:

telnet localhost <porta>

La porta usata dall'emulatore è riportata nel titolo della finestra che lo contiene. In genere è 5554 o una cifra molto simile. Lanciate ora il comando:

geo fix <longitude> <latitude>

Ad esempio:

geo fix 12.484564 41.91247

Fig.2: LocationDemo è un'app che si collega al ricevitore GPS e ne mostra le letture sul display

disponibile.

- **public void onProviderEnabled(String provider)**
  Notifica che il provider indicato in argomento è stato abilitato.
- **public void onProviderDisabled(String provider)**
  Notifica che il provider indicato in argomento è stato disabilitato.
- **public void onLocationChanged(Location location)**
  Notifica la lettura di una nuova posizione.

Una volta che non occorre più conoscere le misurazioni provenienti dal location provider selezionato, è meglio annullare la sottoscrizione svolta in precedenza, agendo sul *LocationManager* con il metodo *removeUpdates()*:

```
locationManager.removeUpdates(myLocationListener);
```

LOCATIONDEMO

Forti delle nozioni appena acquisite, andiamo a realizzare un'applicazione che le metta in pratica. Chiameremo l'applicazione *LocationDemo*. Definiamone il layout nel file *res/layout/main.xml*:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.
                              com/apk/res/android"
 android:orientation="vertical" android:layout_
                                 width="fill_parent"
 android:layout_height="fill_parent">
...
```

Si tratta di un layout molto semplice, che realizza una tabella all'interno della quale andremo ad annotare i dati provenienti dal provider GPS.
Andiamo all'attività *LocationDemoActivity*, che realizza la logica del progetto appena descritta:

```
package it.ioprogrammo.locationdemo;
import java.util.Date;
...
public class LocationDemoActivity extends Activity {
  ...
```

Notate come l'utilizzo del servizio di localizzazione è stato collegato al ciclo di vita dell'attività: si sono utilizzati i metodi *onResume()* e *onPause()*, rispettivamente, per avviare ed interrompere l'utilizzo del servizio. In questa maniera il localizzatore viene invocato solo quando l'attività sta occupando il display, e non quando gira in background.Non resta che mettere tutto insieme in *AndroidManifest.xml*:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/
                                  apk/res/android"
   android:versionCode="1"
   android:versionName="1.0" package=
                      "it.ioprogrammo.locationdemo">
...
```

EMULARE IL GPS

Quando si sviluppa un'applicazione location-based, come l'esempio appena illustrato, ci si scontra con il problema di come eseguirne il test ed il debug. Naturalmente è possibile installare l'applicazione su uno smartphone ed andare a farsi un giro a piedi o in automobile, tuttavia questa non è la maniera migliore per test frequenti e debug efficaci. Per nostra fortuna è possibile impostare delle false coordinate geografiche sull'emulatore fornito con l'SDK. Con Eclipse lo si fa dalla scheda "E*mulator Control*" della prospettiva "*DDMS*". È possibile fornire coordinate semplici, oppure caricarne una lista complessa da file di tipo *GPX* e *KML*. Attenzione: se le coordinate inviate tramite l'Emulator Control di Eclipse DDMS non funzionano correttamente, provate a chiudere l'ambiente e a riavviarlo dopo aver aggiunto le seguenti righe al file *eclipse.ini* presente nella cartelle di installazione del vostro Eclipse:

-Duser.country=US
-Duser.language=en

GOOGLE APIS PER ANDROID

Finora abbiamo appreso come accedere al servizio di localizzazione di Android per conoscere la posizione dell'utente, ma questo è soltanto l'inizio. Una vera e propria applicazione location-based, infatti, fa anche uso di mappe e di altri servizi analoghi. Poiché questo è un terreno in cui Google gioca in casa, abbiamo a nostra disposizione un'ottima libreria di classi per l'accesso facilitato ai servizi location-based di Mountain View. Purtroppo tale libreria, per motivi di licensing, viene fornita separatamente dal sistema, e per questo richiede del lavoro preparatorio aggiuntivo. Per prima cosa è necessario aver scaricato e integrato la libreria nel proprio Android SDK. Potete verificarlo avviando il manager dei dispositivi virtuali e controllando che nell'elenco dei pacchetti installati ("Installed Packages") sia presente un pacchetto del tipo "Google APIs by Google Inc.", associato alla versione di Android per la quale state sviluppando. Se non lo trovate, muovetevi nella scheda "Available Packages" e scaricatelo. Ad operazione completata create un dispositivo virtuale che supporti le Google API appena installate (lo potete fare dalla scheda "Virtual Devices"). In Eclipse, quando creerete una nuova applicazione che usa le mappe di Google, ricordatevi di indicare esplicitamente l'utilizzo della libreria nel target del progetto. L'ambiente, creando il nuovo progetto, aggiungerà la libreria al build path dell'applicazione. Se non utilizzate Eclipse dovrete

svolgere questo compito a mano. I JAR delle Google API li trovate a partire dal percorso *<android-sdk-directory>/add-ons*. Accertatevi, infine, che nel manifest della vostre applicazioni location-based siano comprese la dichiarazione di utilizzo della libreria Google Maps e le clausole di utilizzo dei permessi per l'accesso al location manager e ad Internet. Solitamente lo scheletro del manifest di un'app location-based assomiglia al seguente:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest ...>
  <application ...>
    <uses-library android:name="com.google.android.
                                              maps" />
    ...
  </application>
  <uses-permission android:name="android.permission.
                           ACCESS_FINE_LOCATION" />
  <uses-permission android:name="android.permission.
                         ACCESS_COARSE_LOCATION" />
  <uses-permission android:name="android.permission.
                                         INTERNET"/>
</manifest>
```

GOOGLE MAPS KEY

Ancora un passo ci separa dal poter mostrare mappe sul display del nostro dispositivo: dobbiamo ottenere una chiave (*key*) per l'accesso al servizio Google Maps. Per farlo dobbiamo fornire a Google l'impronta MD5 del certificato che utilizziamo per firmare le nostre applicazioni. Siccome solitamente si utilizza un certificato per lo sviluppo ed uno per il rilascio, sarà probabilmente necessario ottenere ed utilizzare due chiavi differenti. Il certificato per lo sviluppo ed il debug è compreso nel keystore che trovate ad uno dei seguenti percorsi:

- Windows Vista: *C:\Users\<username>\.android\debug.keystore*
- Windows XP: *C:\Documents and Settings\<username>\.android\debug.keystore*
- MacOS X e Linux: *~/.android/debug.keystore*

In Eclipse potete verificare il percorso del keystore di debug aprendo le preferenze dell'ambiente ed entrando nella voce "*Android » Build*".
L'impronta MD5 di un certificato può essere calcolata usando l'utility *keytool*, compresa in ogni Java SDK. Con il prompt dei comandi posizionatevi all'interno della directory *bin* del vostro JDK, in modo che il comando *keytool* sia a portata di lancio. Quindi eseguite il comando:

```
keytool -keystore <percorso-keystore> -list
```

Vi sarà richiesta la password del keystore. La password di default del keystore di debug è android. Adesso *keytool* vi mostrerà l'elenco dei certificati compresi nel keystore, e per ognuno di essi vi fornirà l'impronta MD5. Con il browser, recatevi ora all'indirizzo: http://code.google.com/android/add-ons/google-apis/maps-api-signup.html

Accettate le condizioni proposte e inserite l'MD5 del certificato per il quale volete ottenere una chiave di utilizzo di Google Maps. È fatta! Salvate la pagina Web con la chiave generata per il vostro certificato.

MAPACTIVITY E MAPVIEW

Il pacchetto di riferimento per le API Android di Google Maps è *com.google.android.maps*. All'interno vi trovate *MapActivity*, che è la classe da estendere nel caso in cui si voglia realizzare un'attività che mostri delle mappe sullo schermo. Estendere *MapActivity* è esattamente come estendere *Activity*, con l'unica differenza che si deve implementare il metodo:

```
protected boolean isRouteDisplayed()
```

A questo metodo dobbiamo far restituire *true* quando si sta visualizzando una mappa con delle indicazioni stradali (ad esempio un percorso da un punto *A* ad un punto *B*). In tutti gli altri casi si deve restituire *false*.
L'altra differenza tra *Activity* e *MapActivity* è che, all'interno di quest'ultima, è possibile introdurre un widget di tipo *MapView*. Si tratta del componente in grado di mostrare le mappe provenienti dai server di Google. Potete dichiarare il componente di tipo *MapView* in un layout XML, da caricare poi all'interno della *MapActivity*:

```
<com.google.android.maps.MapView
  android:apiKey="CHIAVE GOOGLE MAPS QUI"
  android:id="@+id/mapView"
  android:layout_width="fill_parent"
  android:layout_height="fill_parent" />
```

Il widget potrà poi essere recuperato dall'interno della *MapActivity*, facendo:

```
MapView mapView = (MapView) findViewById(R.
```

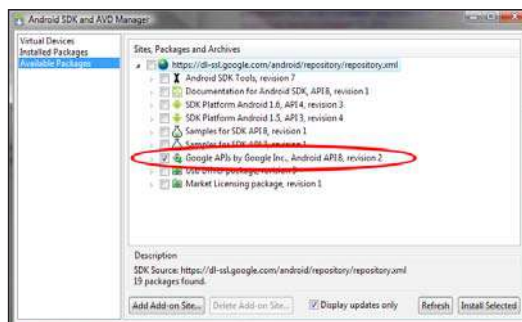



Fig.3: Le API di Google per la gestione delle mappe vanno scaricate ed installate separatamente dal sistema

GOOGLE MAPS API REFERENCE

La reference guide per le Google Maps API di Android è all'indirizzo: http://code.google.com/android/add-ons/google-apis/reference/index.html

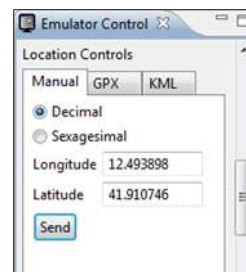



Fig.4: Con l'Emulator Control di Eclipse DDMS è possibile inviare all'emulatore false coordinate di localizzazione

Fig.5: L'utilizzo delle Google API va indicato come target del nuovo progetto Android che si sta creando in Eclipse

id.mapView);
A questo punto i metodi di *MapView* possono essere invocati per controllare la mappa mostrata. Diamo uno sguardo ai metodi di più comune utilizzo:

- **public void setSatellite(boolean on)**
  Attiva o disattiva la vista da satellite.
- **public boolean isSatellite()**
  Controlla se la vista da satellite è attiva oppure no.
- **public void setStreetView(boolean on)**
  Attiva o disattiva i tracciati che mostrano dove street view è disponibile.
- **public void setClickable(boolean on)**
  Rende la mappa cliccabile oppure no. Se la mappa è cliccabile (e per default non lo è), l'utente può controllarla con il tocco del dito.
- **public void setBuiltInZoomControls(boolean on)**
  Attiva o disattiva i controlli incorporati per lo zoom della mappa.
- **public int getZoomLevel()**
  Restituisce il livello di zoom corrente, che sarà sempre compreso tra 1 e 21.
- **public GeoPoint getMapCenter()**
  Restituisce le coordinate geografiche del punto centrale della mappa.

Gli oggetti *GeoPoint*, come è facile intuire, sono utilizzati per esprimere una posizione terrestre. Il solo costruttore disponibile è: *public GeoPoint(int latitudeE6, int longitudeE6)* Latitudine e longitudine, in questo caso, sono espressi mediante degli interi. Nel caso del *LocationManager*, come abbiamo visto in precedenza, sono invece espressi con valori double. Passare da una notazione all'altra è molto semplice:

```
int asInt = (int) Math.floor(asDouble * 1.0E6);
```

Le coordinate contenute in un oggetto *GeoPoint* possono essere recuperate con i metodi *getLatitudeE6()* e *getLongitudeE6()*. Per passare dalla notazione intera a quella decimale si può fare:

```
double asDouble = asInt / 1.0E6;
```

EVENTI DI TOUCH SUGLI OVERLAY

Gli overlay possono essere interattivi. Per riscontrare eventuali eventi di tocco o di tastiera riscontrati su di essi, si possono ridefinire i metodi di *Overlay* *onTap()*, *onTouchEvent()*, *onKeyDown()* e *onKeyUp()*.

MAPDEMO

Realizziamo insieme un primo esperimento con le mappe. Chiameremo l'applicazione *MapDemo*. Ecco il suo layout da conservare sul file *res/layout/main.xml*:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.
                                        com/apk/res/android"
  android:orientation="vertical" android:layout_
                                        width="fill_parent"
  android:layout_height="fill_parent">
...
```

Questo file realizza un layout dove la mappa di Google è il componente principale, ed in alto sono posti tre checkbox: "*Satellite*", "*Traffic*" e "*Street View*". Li utilizzeremo per consentire all'utente di abilitare o disabilitare le corrispondenti caratteristiche. Passiamo ora all'attività *MapDemoActivity*:

```
package it.ioprogrammo.mapdemo;
...
import com.google.android.maps.MapActivity;
import com.google.android.maps.MapView;
...
public class MapDemoActivity extends MapActivity {
...
```

Qui, con veramente poche righe di codice, si riesce a collegare i tre checkbox di cui sopra al widget *MapView*. Manca solo l'*AndroidManifest.xml*:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/
                                        apk/res/android"

  package="it.ioprogrammo.mapdemo"
                                        android:versionCode="1"
...
```

CONTROLLARE LA MAPPA

Rendendo un widget *MapView* cliccabile e abilitando i controlli per lo zoom compresi nel widget, l'utente può scorrere la mappa e controllare ciò che più lo aggrada. Per controllare la mappa in maniera automatica, invece, è possibile farsi restituire da *MapView* un oggetto di tipo *MapController*, grazie al metodo *getController()*: *MapController mapController = mapView.getController();* Con il *MapController* si possono fare diverse cose, tra cui:

- **public boolean zoomIn()**
  Aumenta lo zoom di un livello. Restituisce *true* se l'operazione riesce, *false* se il livello di zoom è già al massimo.
- **public boolean zoomOut()**
  Diminuisce lo zoom di un livello. Restituisce *true* se l'operazione riesce, *false* se il livello di zoom è già al massimo.
- **public int setZoom(int zoomLevel)**
  Imposta il livello di zoom della mappa, che va espresso come valore compreso tra 1 e 21. Restituisce il livello di zoom effettivamente impostato.
- **public void setCenter(GeoPoint point)**
  Sposta la mappa in modo che il punto indicato sia al centro.
- **public void animateTo(GeoPoint point)**
  Fa muovere la mappa, mediante un'animazione, fino al punto indicato.

Fig.6: La chiave per l'utilizzo delle Google Maps API va richiesta online fornendo l'impronta MD5 del certificato che sarà usato per firmare l'applicazione Android

Ora che siamo in grado di manipolare la mappa mostrata, proviamo a collegare l'esempio del paragrafo precedente con il servizio di localizzazione studiato nella prima parte dell'articolo. Facciamo in modo che l'attività, una volta avviata, inizi ad ascoltare le coordinate provenienti dal ricevitore GPS, centrando di volta in volta la mappa sulla posizione letta. Incrementiamo anche il valore di zoom di partenza, in modo da evidenziare meglio la posizione dell'utente. Modifichiamo l'attività *MapDemoActivity*, aggiungendole quanto indicato nel seguente stralcio:

```
...
import com.google.android.maps.GeoPoint;
import com.google.android.maps.MapController;
public class MapDemoActivity extends MapActivity {
...
```

Completiamo l'opera aggiungendo l'uso del permesso *android.permission.ACCESS_FINE_LOCATION* nel manifest dell'applicazione.

OVERLAY

Quando un'applicazione mostra una mappa, il più delle volte è perché vuole indicare qualcosa, magari attraverso un disegno sovrapposto alla mappa stessa. Ad esempio, si è soliti inserire una freccia che indica la posizione dell'utente e dei baloon per evidenziare i luoghi di suo interesse che sono nei paraggi. In gergo si dice che alla mappa vengono sovrapposti degli *overlay*, ognuno dei quali mostra qualcosa in particolare. Gli overlay sovrapposti ad un widget *MapView* possono essere recuperati e controllati grazie al metodo:

```
public java.util.List<Overlay> getOverlays()
```

Un nuovo overlay può essere aggiunto alla lista facendo:

```
mapView.getOverlays().add(new MyOverlay());
```

Per realizzare un overlay personalizzato si deve estendere la classe *Overlay*. Il metodo che più comunemente viene ridefinito è:

public void draw(android.graphics.Canvas canvas, MapView mapView, boolean shadow)

Questo metodo viene invocato automaticamente per richiedere il disegno dell'overlay. L'argomento *canvas* è un pannello sul quale è possibile fare disegno libero (ce ne occuperemo più nel dettaglio in un episodio futuro), *mapView* è la mappa di riferimento e *shadow*, infine, indica se si deve disegnare o meno l'ombra dell'overlay che si sta realizzando. Il metodo viene sempre chiamato due volte: la prima volta con *shadow* pari a *true*, per disegnare l'ombra, la seconda con *shadow* pari a *false*, per disegnare il contenuto di primo piano dell'overlay. Se non si è interessati all'ombra, è sufficiente non far nulla quando *shadow* è uguale a *true*. Il problema che sorge, a questo punto, è che sugli oggetti *Canvas* si disegna ragionando in termini di coordinate x ed y, mentre le mappe ed i servizi di localizzazione lavorano con le coordinate geografiche latitudine e longitudine. Come fare, quindi, per disegnare qualcosa alla coordinata (*x, y*) che corrisponde esattamente ad una posizione (*lat, lon*) della mappa? Si può passare da una notazione all'altra estraendo un oggetto *Projection* dal *MapView* in uso:

Projection projection = mapView.getProjection();

Con un oggetto *Projection* si può passare da coordinate geografiche a coordinate di disegno, grazie al metodo:

public android.graphics.Point toPixels(GeoPoint in, android.graphics.Point out)

Le coordinate del *GeoPoint* vengono convertite e salvate in un oggetto *Point*, che contiene delle semplici coordinate *x* ed *y*. Si può eseguire anche la conversione inversa, grazie al metodo:

public GeoPoint fromPixels(int x, int y)

Andiamo ad apportare l'ultima miglioria alla nostra applicazione *MapDemo*. Aggiungiamo fra le sue risorse un'immagine *youarehere.png* (la trovate nel codice allegato), che rappresenta una freccia puntata verso il basso. Va disposta al percorso di progetto *res/drawable/youarehere.png*. La useremo in un overlay che indicherà all'utente dove si trova. Modifichiamo ancora una volta l'attività *MapDemoActivity*, con le aggiunge mostrate di seguito:

```
package it.ioprogrammo.mapdemo;
...
  private class CurrentPositionOverlay extends Overlay {
...
```

Non resta che caricare l'applicazione su un dispositivo Android ed andarsi a fare un giro per la propria città. Buona passeggiata!

Carlo Pelliccia

Fig.7: La posizione indicata dal ricevitore GPS viene evidenziata sulla mappa grazie ad una freccia disegnata in un overlay della mappa

Carlo Pelliccia lavora presso 4IT (www.4it.it), dove si occupa di analisi e sviluppo software per piattaforme Java. Nella sua carriera di technical writer ha pubblicato cinque manuali ed oltre duecento articoli, molti dei quali proprio tra le pagine di ioProgrammo. Il suo sito, che ospita anche diversi progetti Java Open Source, è disponibile all'indirizzo www.sauronsoftware.it

APP ANDROID FACILI CON APP INVENTOR

APP INVENTOR È IL NUOVO SISTEMA DI GOOGLE PER CREARE APPLICAZIONI ANDROID SENZA SCRIVERE UNA SOLA RIGA DI CODICE. SCOPRIAMO IN COSA CONSISTE E UTILIZZIAMOLO PER REALIZZARE FACILMENTE LE NOSTRE IDEE

Le applicazioni contano. Questa è la conclusione alla quale sono finalmente giunti tutti i produttori di smartphone di ultima generazione. Non basta più proporre un hardware potente, un design accattivante e un sistema facile da utilizzare: ci vuole anche un ampio parco di applicazioni, possibilmente di alta qualità. Non si fanno discriminazioni per genere, utilità o complessità: l'utente moderno, sul market del suo smartphone, vuole trovare di tutto: dallo strumento di lavoro al passatempo, dall'applicazione per interagire con il suo social network preferito a quella utile per risolvere una piccola faccenda di tutti i giorni.
Google tutto ciò l'ha capito benissimo e per questo la mamma di Android coccola ed incentiva chiunque abbia nuove idee per applicazioni di successo.
Gli sviluppatori che hanno scelto Android, infatti, hanno vita facile: hanno a disposizione un linguaggio di programmazione facile ed espressivo, una libreria di funzionalità potenti e complete, un ambiente di sviluppo aperto e flessibile. La documentazione e gli articoli tecnici non mancano, così come abbondanti sono le comunità online dove gli sviluppatori Android si radunano e si danno aiuto a vicenda. Programmare applicazioni per Android, insomma, è facile e divertente. A Google, però, questo non basta. La programmazione, infatti, è molto importante per avere buone applicazioni, e per questo va incentivata, ma allo stesso tempo la necessità di scrivere del codice è un limite per chi non ha studiato informatica. Insomma, per quanto facile e divertente possa essere lo sviluppo delle applicazioni Android, la capacità di scrivere del buon codice resta sempre appannaggio degli addetti ai lavori o di chi studia per diventarlo.
Per questo motivo Google sta gradualmente introducendo *App Inventor*, un sistema alternativo per creare applicazioni Android senza scrivere una sola riga di codice. Grazie ad *App Inventor* le applicazioni possono essere letteralmente disegnate sullo schermo del proprio computer. Apprendendo alcuni principi di base, tra l'altro molto semplici, si diventa velocemente in grado di creare, verificare e vendere (o regalare) delle applicazioni Android competitive. Insomma, basta l'idea.

PREREQUISITI

App Inventor è un'applicazione web online, come GMail o Facebook per intenderci, e perciò per accedere all'applicazione bastano un computer ed un browser di comune fattura. Più nello specifico, i sistemi attualmente supportati sono:

- Windows: XP, Vista, 7
- Mac OS X: 10.5, 10.6
- GNU/Linux: Ubuntu 8+, Debian 5+

I browser compatibili, invece, sono:

- Mozilla Firefox 3.6 o successivo
- Apple Safari 5.0 o successivo
- Google Chrome 4.0 o successivo
- Microsoft Internet Explorer 7 o successivo

È poi importante che il sistema ed il browser installino Java 6. Per verificare ed eventualmente soddisfare questo ulteriore requisito, è sufficiente collegarsi a questa pagina www.java.com. Per verificare il vostro sistema seguite il link "*Io ho Java?*". In caso di esito negativo, procedete al download e all'installazione dal sito stesso.

SETUP DEL SOFTWARE

Benché l'App Inventor in sé, essendo un'applicazione online, non necessiti dell'installazione di ulteriore software per lo sviluppo delle applicazioni Android, conviene comunque scaricare ed installare un package aggiuntivo, chiamato *App Inventor Setup Software.* Il pacchetto contiene degli strumenti aggiuntivi per la verifica ed il confezionamento delle applicazioni e serve soprattutto per verificare le applicazioni realizzate su un dispositi-

vo reale connesso via USB al computer, oppure in alternativa su un emulatore.
A seconda del vostro sistema operativo, collegatevi ad uno fra i seguenti indirizzi:

Windows:
http://appinventor.googlelabs.com/learn/setup/setupwindows.html

GNU/Linux:
http://appinventor.googlelabs.com/learn/setup/setuplinux.html

Mac OS X:
http://appinventor.googlelabs.com/learn/setup/setupmac.html

Scaricate ed installate il pacchetto, secondo le istruzioni riportate nella pagina. Per verificare che l'installazione sia andata a buon fine, recatevi nella directory in cui è stato installato il software ed avviate il comando *run-emulator*. Come il nome lascia intuire, questo comando avvia l'emulatore del sistema Android compreso nel pacchetto. Se tutto è andato a buon fine, dopo qualche minuto di attesa, dovreste ottenere sul vostro desktop l'emulazione della più recente piattaforma Android, come mostrato in Fig.1 Prendete inoltre nota del percorso di installazione del software appena configurato: più tardi questa informazione potrebbe tornare utile.

Fig. 1: L'emulatore Android compreso nell'App Inventor Setup Software

CREARE UN ACCOUNT

Soddisfatti i requisiti hardware e software, bisogna ottenere l'accesso all'applicazione online. Come si diceva sopra, infatti, App Inventor è come Facebook o GMail. Per accedervi bisogna avere un account e chiedere l'abilitazione all'uso del servizio. Collegatevi quindi all'indirizzo: appinventor.googlelabs.com
Se avete già un account Google potete accedere direttamente (box evidenziato in blu in Fig.2), altrimenti seguite il link per iscrivervi (box evidenziato in rosso). Dopo aver soddisfatto il requisito dell'account e dell'accesso ai servizi Google, bisogna compilare un ulteriore modulo per richiedere l'accesso alla piattaforma (link evidenziato in verde).

Fig. 2: La pagina di accesso ad App Inventor

App Inventor, infatti, è una tecnologia ancora in beta, e perciò non è ancora aperta all'utilizzo da parte del grande pubblico. È tradizione di Google, infatti, sperimentare le nuove soluzioni con una cerchia ristretta di utenti, per renderle poi pubbliche successivamente. Anche con GMail ed altre celebri applicazioni online si fece in questa maniera. Compilate dunque il modulo di richiesta (l'unico dato effettivamente indispensabile che dovete fornire è il vostro indirizzo *@gmail.com*), inviatelo con il tasto "*Submit*" ed attendete fiduciosi. Solitamente non bisogna attendere a lungo: in alcuni casi fortunati poche ore, in altri qualche giorno. Controllate comunque regolarmente la casella di posta che avete segnalato, visto che è lì che riceverete la notifica di avvenuta attivazione del servizio per il vostro account. Non appena sarete abilitati, tornate all'indirizzo dell'App Inventor riportato sopra ed accedete finalmente alla piattaforma.

CREARE UN NUOVO PROGETTO

Iniziamo ad imparare come funziona App Inventor. La piattaforma è organizzata per progetti. Ogni applicazione Android che vogliamo realizzare fa parte di un progetto differente. Per questo, non appena si accede ad App Inventor, la prima cosa che viene visualizzata è l'elenco dei progetti sui

SAVE E SAVE AS

Il bottone "*Save*" presente nell'interfaccia di App Inventor permette di salvare, di tanto in tanto, il lavoro svolto. Benché la piattaforma svolga comunque dei salvataggi automatici, vi consigliamo di utilizzarlo ogni volta che avete concluso un lotto di modifiche. Il tasto "*Save As*", invece, permette di salvare il lavoro attuale in un altro progetto, diverso da quello di partenza.

Fig. 3: La pagina principale di App Inventor, con l'elenco dei progetti inizialmente vuoto

quali si sta lavorando. Appena ci si è iscritti, naturalmente, non si sta lavorando ancora su alcun progetto, e quindi sarà anzitutto necessario crearne uno. Azionate il bottone "*New*". Vi sarà chiesto di assegnare un nome al nuovo progetto. I nomi dei progetti devono essere semplici: possono contenere solo lettere, numeri e caratteri underscore, e non possono inglobare spazi o altri caratteri speciali. Digitiamo il nome di progetto "prima_applicazione" e confermiamo. Il nuovo progetto sarà creato ed aperto automaticamente. La schermata presentata a questo punto da App Inventor è il principale banco di lavoro della piattaforma, nel quale è possibile costruire l'applicazione. Al centro dello schermo c'è l'anteprima di come si presenta il software in lavorazione. Sulla sinistra c'è invece la palette dei componenti. Un *componente*, nel gergo di App Inventor, è un elemento di base che è possibile impiegare nella costruzione dell'applicazione, come fosse un mattone. Esempi di componenti sono i bottoni, le immagini, le etichette con dei messaggi di testo e così via. Le applicazioni vengono costruite combinando insieme una serie di componenti. La palette dei componenti è suddivisa in differenti categorie, che comprendono un po' di tutto. Per capirci meglio: tra i componenti di base troverete i bottoni, le etichette e le immagini citati prima, mentre tra quelli più avanzati troverete cose come i riproduttori di suoni e di video, o i componenti per interagire con l'accelerometro ed il ricevitore GPS del telefono.

Fig. 4: La maschera per la creazione di un nuovo progetto

Fig. 5: Il nuovo progetto creato

NOTA

CHECKPOINT

Il bottone "*checkpoint*" salva un'istantanea dello stato attuale del progetto in modo tale che, successivamente, sia possibile recuperarla e ripristinare così una situazione antecedente. Immaginate di voler provare ad applicare al vostro progetto una modifica di cui non siete sicuri. Create un checkpoint del progetto ed applicate poi la modifica: se deciderete di annullarla, potrete tornare a lavorare sulla versione antecedente così come era al momento del checkpoint.

DESIGN DI UN'APPLICAZIONE

Per prendere confidenza con App Inventor andiamo a realizzare insieme un primo semplice ma valido esempio. Realizzeremo un'applicazione che, una volta lanciata, mostri l'immagine di un'automobile. Sotto questa immagine ci sarà un pulsante, con il messaggio "*Metti in moto*". Quando l'utente premerà il bottone, il telefono dovrà riprodurre il suono del rombo di un motore. Grazie ad App Inventor siamo in grado di farlo in pochissimi minuti.

Per prima cosa ci serve il materiale di base, cioè un'immagine formato JPG di un'automobile ed un file MP3 con il rombo del motore. Potete cercarli su Internet, oppure utilizzare quelli compresi nel CD-Rom allegato alla rivista.

Costruiamo l'interfaccia grafica di cui abbiamo bisogno. Dalla palette, nel gruppo "*Basic*", selezioniamo il componente "*Image*" (immagine) e trasciniamolo con il mouse all'interno dell'applicazione.

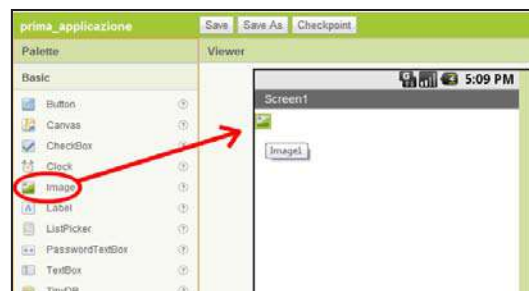



Fig. 6: Per aggiungere un componente all'applicazione è sufficiente trascinarlo dalla palette all'anteprima

Trasciniamo anche il componente "*Button*" (bottone), posizionandolo subito sotto l'immagine.

Nella palette dei componenti localizziamo ora il componente "*Sound*", compreso nel gruppo "*Media*". Trasciniamo anche questo all'interno della applicazione. A differenza degli altri due componenti, "*Sound*" è invisibile, non ha cioè elementi di interfaccia. Non stupitevi, pertanto, se dopo averlo trascinato non noterete alcuna differenza nell'anteprima dell'applicazione.

Dopo aver popolato l'applicazione con tutti i componenti necessari, diamo un'occhiata sulla destra. Ci sono tre aree molto importanti:

- **Components**. In quest'area vengono ripor-

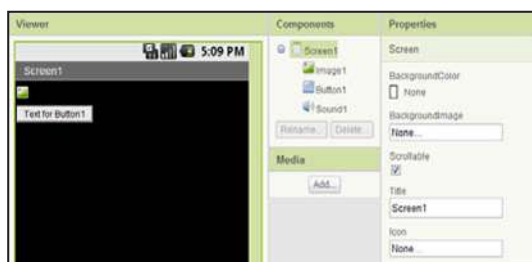



Fig. 7: Le aree "Components", "Media" e "Properties" permettono di gestire ciascun componente presente nell'applicazione

tati tutti i componenti usati nell'applicazione.

- Il componente di base, che si chiama "*Screen1*", rappresenta la finestra principale dell'applicazione. Al suo interno ci sono tutti i componenti trascinati nella schermata (nel nostro caso saranno "*Image1*", "*Button1*" e "*Sound1*"). Selezionando uno dei componenti è possibile rinominarlo (tasto "*Rename*") o eliminarlo definitivamente dalla schermata ("*Delete*").

- **Properties**. Contiene gli strumenti utili per modificare le proprietà di ciascun elemento dell'applicazione. Per modificare l'aspetto di un componente è anzitutto necessario selezionarlo nell'anteprima o dall'elenco presente nell'area "*Components*". Fatto ciò, la scheda "*Properties*" riporterà tutti i dettagli del componente selezionato che è possibile modificare. Quali siano le proprietà modificabili dipende dalla natura del componente.

- **Media**. Qui vengono riportati tutti i file multimediali (immagini, audio, video) necessari all'applicazione. Con il tasto "*Add*" è possibile avviare la procedura di upload di un nuovo file.

Utilizziamo questi strumenti. Per prima cosa, usiamo l'area "*Media*" per caricare l'immagine e l'audio di cui si è detto in precedenza.
Andiamo poi su "*Components*". Selezioniamo "*Screen1*", che corrisponde alla finestra principale dell'applicazione, e modifichiamo alcune sue caratteristiche nell'area "*Properties*". Cambiamo il colore di sfondo impostandolo su nero, e modifichiamo il titolo della finestra in modo che diventi "*Automobile*".
Passiamo ora a "*Image1*". Impostiamo l'immagine da mostrare, alterando la proprietà "*Picture*".
Associamogli l'immagine dell'automobile caricata poco prima nella sezione "*Media*". Modifichiamo anche le proprietà del componente "*Button1*".
In particolare, cambiamo il testo contenuto al suo interno, impostando la proprietà "*Text*" sul valore "*Metti in moto*". Finiamo modificando anche "*Sound1*". Non dobbiamo far altro che associargli l'MP3 caricato in precedenza.

BLOCKS EDITOR

L'interfaccia dell'applicazione, a questo punto, è pronta: non ci sono né altri componenti da aggiungere né altri file multimediali da caricare. Manca però qualcosa. Dobbiamo ancora fare in modo che quando l'utente aziona il bottone, il rombo del motore venga effettivamente riprodotto. Ci manca, insomma, quello che è il cuore di ogni applicazione: la logica, cioè la sequenza di indicazioni di tipo causa-effetto che sono il cuore di ogni applicazione. Descrivere una logica è ciò per cui i linguaggi di programmazione sono nati. Il codice serve proprio per gestire il legame tra le cause e gli effetti, ed infatti ogni codice suona sempre come «*se accade questo, allora fai quest'altro*». Come è possibile, quindi, stabilire queste sequenze di azioni di questo genere se App Inventor, come si è detto in apertura, non richiede la conoscenza di un linguaggio di programmazione?
La risposta è: attraverso il *Blocks Editor*. Si tratta di un editor visuale che sostituisce completamente la necessità di scrivere codice, e lo fa attraverso delle rappresentazioni grafiche dei flussi causa-effetto. Spiegarlo è molto più difficile che provarlo, andiamo dunque a sperimentarlo in prima persona.
In alto a destra, l'App Inventor presenta il tasto "*Open the Blocks Editor*". Premiamolo.
Il Blocks Editor è un applicativo esterno realizzato in Java. Premendo sul tasto, il software sarà scaricato ed eseguito. Per questo è molto probabile che vi venga richiesto il consenso per l'esecuzione. Naturalmente bisogna acconsentire. Alla prima esecuzione, inoltre, il Blocks Editor vi chiederà probabilmente di digitare il percorso del comando *adb*, che è una delle utilità installate in precedenza con il pacchetto App Inventor Setup Software, e che serve per connettersi all'emulatore o ad un dispositivo reale. Andate quindi sul vostro disco a trovare il percorso del comando *adb.exe* (Windows) o *adb* (Linux e Mac). Fornite tale percorso in maniera completa, ad esempio (Windows):

Fig. 8: L'anteprima dell'interfaccia realizzata

SCARICARE I SORGENTI

Volete condividere con altre persone il sorgente delle vostre applicazioni realizzate con l'App Inventor? Andate nella lista dei progetti, selezionate uno, quindi dalla lista "*More actions*" selezionate la voce "*Download Source*". Scaricherete così un archivio ZIP che contiene il progetto selezionato. Un'altra persona potrà importarlo nel suo account su App Inventor, selezionando questa volta la voce "*Upload Source*".
In questa maniera potrà anche lui lavorare sul vostro progetto.

```
C:\Program Files\AppInventor\commands-for-
                    Appinventor\adb.exe
```

Una volta avviato, il Blocks Editor si presenta come in Fig.9

Fig. 9: Il Blocks Editor, cioè lo strumento per definire la logica di funzionamento dell'applicazione

ESPORTARE APK

La forma finale per la distribuzione delle applicazioni Android completate consiste in un file con estensione *.apk*. Avendo a disposizione un file APK è possibile distribuire l'applicazione sul market o attraverso altri canali. Per ottenere l'APK di una vostra applicazione ormai completa, aprite il progetto corrispondete su App Inventor e dalla schermata di lavoro principale aprite l'elenco "*Package for Phone*" e scegliete la voce "*Download to this Computer*". Potrete così scaricare il file *.apk* che contiene la vostra applicazione finita.

All'interno del Blocks Editor possiamo ritrovare tutti i componenti che abbiamo inserito all'interno dell'applicazione. Sono nella scheda corrispondente alla linguetta "*My Blocks*".

Fig. 10: I blocchi disponibili per il componente "Button1"

Tocchiamo la voce "*Button 1*": apparirà un elenco pieno di blocchi che corrispondono ad eventi e proprietà del componente. È subito evidente che i blocchi mostrati sono raggruppabili in categorie differenti.
I blocchi verdi, che chiameremo eventi, sono tutti del tipo «*when ... do*», ad esempio «*when Button1.Click do*», che significa «*quando viene cliccato Button1 fai questo*». Al loro interno, infatti, c'è spazio per inserire un blocco di tipo azione, come quelli del tipo «*call ...*» e «*set ... to*».
I blocchi *call* eseguono un'azione che il componente può svolgere, ad esempio «*call Sound1.Play*» esegue il suono contenuto nel componente "*Sound1*".

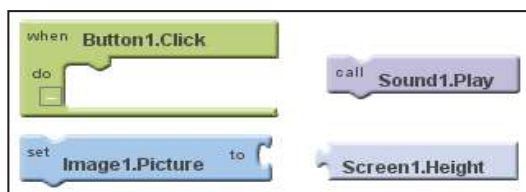



Fig. 11: Differenti tipologie di blocchi a confronto

I blocchi *set*, invece, cambiano il valore di una proprietà di un componente, ad esempio «*set Image1. Picture to*» cambia l'immagine mostrata dal componente "*Picture1*".
I blocchi *set* vanno collegati ad un'altra categoria di blocchi, che fornisce il nuovo valore da impostare. Questo valore può essere letto dalle proprietà di un altro componente (quarto tipo di blocco in figura), oppure specificato in altro modo attraverso una delle altre caratteristiche messe a disposizione dal Blocks Editor (provate ad esplorare le categorie di blocchi messe a disposizione nella linguetta "*Built-in*"). Facciamo ora in modo che alla pressione del tasto "*Button1*" il suono del componente "*Sound1*" venga riprodotto.
Trasciniamo allora il blocco «*when Button1.Click do*» nell'area di lavoro. Colleghiamolo quindi con il blocco «*call Sound1.Play*», come mostrato in Fig.12.
Proviamo a fare di più, aggiungendo una seconda conseguenza alla pressione del bottone. Facciamo che, dopo aver avviato la riproduzione di "*Sound1*", il testo di "*Button1*" cambi da "*Metti in moto*" a "Vroooooam!".
È molto semplice farlo: aggiungete il blocco «*set Button1.Text to*» all'evento *Button1.Click*. Il nuovo testo può essere specificato usando uno dei blocchi

Fig. 12: La logica della nostra prima applicazione

del gruppo "Built-in".
Trascinate il blocco text nell'area ed agganciatelo a «*set Button1.Text to*».
A questo punto, cliccando sul nuovo blocco, potrete digitare il testo da impostare. Provate ad esempio digitando "*Vroooooam*!", oppure la variante che preferite.

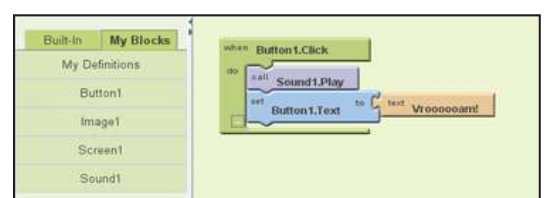



Fig. 13: Ora sono state impostate due conseguenze al clic del tasto "Button1"

ESEGUIRE L'APPLICAZIONE

Scommetto che non vedere l'ora di provare l'applicazione che abbiamo appena realizzato. Possiamo farlo con l'emulatore o con un dispositivo reale.
Per utilizzare un dispositivo reale è necessario collegarlo al computer via USB, non prima di aver installato gli eventuali driver forniti dal produttore.

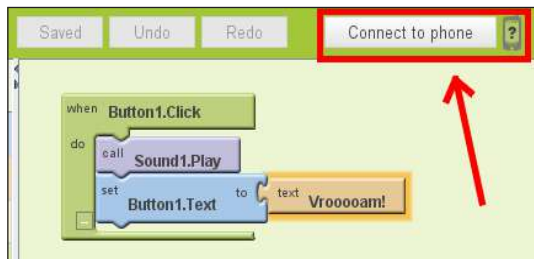



***Fig. 14:** Il tasto "Connect to phone" permette l'esecuzione dell'applicazione su un dispositivo reale o su un emulatore*

Per usare l'emulatore, invece, è sufficiente lanciare il comando *run-emulator* dimostrato in precedenza e aspettare che il sistema emulato venga caricato.
Dall'interno del Blocks Editor, in alto, azionate il tasto "*Connect to phone*".
Attendete qualche istante: il dispositivo (reale o emulato che sia) sarà contattato dall'App Inventor, e l'applicazione sarà scaricata ed eseguita al suo interno.

PROGETTO LAVAGNA

Realizziamo un secondo progetto con App Inventor. Cimentiamoci con qualcosa di leggermente più complicato (ma, come vedremo, sempre semplicissimo). Sfruttiamo il touchscreen del nostro dispositivo Android, realizzando un'applicazione che funzioni un po' come una lavagna. L'utente, con il dito, potrà tracciare linee e segni sullo schermo. In più, potrà stabilire il colore del gesso virtuale: bianco, giallo o rosso. Per completare l'opera, mettiamo a disposizione anche un tasto che funzioni come un cancellino, ripulendo completamente la superficie dello schermo-lavagna. Torniamo alla lista dei progetti di App Inventor (link "*My Projects*"). Da qui creiamo un nuovo progetto, che chiameremo "*lavagna*". Costruiamo quindi un'interfaccia come quella mostrata in Fig.15. Per realizzare tale interfaccia, i passi da seguire sono:

***Fig. 15:** L'applicazione in esecuzione sull'emulatore*

1. Personalizziamo "Screen1", cambiando il titolo della finestra in "Lavagna" e cambiando in nero il colore dello sfondo.
2. Trasciniamo dalla palette un componente "*HorizontalArrangement*", che è parte dell'elenco "*Screen Arrangement*". Questo componente permette di disporre in sequenza orizzontale una serie di altri componenti.
3. Dentro l'elemento "*HorizontalArrangement*" appena disposto, inseriamo una serie di quattro componenti "Button".
4. Personalizziamo i bottoni. Cambiamo anzitutto i loro nomi da "*Button1*", "*Button2*", "*Button3*" e "*Button4*" a "*ButtonBianco*", "*ButtonGiallo*", "*ButtonRosso*" e "*ButtonPulisci*". Cambiamo il testo dei bottoni in modo che sia: "*Bianco*", "*Giallo*", "*Rosso*" e "*Pulisci*". Cambiamo inoltre l'aspetto dei pulsanti: mettiamo su nero il colore di sfondo di tutti e quattro i bottoni, aumentiamo le dimensioni del testo (valore 20), impostiamo l'uso del grassetto e cambiamo il colore del testo in modo che ogni bottone esponga quello più appropriato. Mettiamo infine in corsivo la scritta "Pulisci" sul quarto bottone.
5. Trasciniamo dalla palette un componente di tipo "*Canvas*". Posizioniamolo subito sotto l'elemento "*HorizontalArrangement*" disposto in precedenza. I componenti di tipo "*Canvas*" permettono di tracciare linee liberamente sullo schermo: proprio quello che ci serve! Maneggiamo le proprietà dell'oggetto "*Canvas*": mettiamo lo sfondo sul colore nero ed il colore di disegno ("*PaintColor*") impostiamolo sul bianco. Infine aggiustiamo le dimensioni dell'elemento: per la larghezza scegliamo lo speciale valore "*fill_parent*" (rende l'elemento largo quanto tutto lo schermo) e per l'altezza digitiamo il valore preciso di 370 pixel. Accertiamoci infine che il nome del componente sia proprio "*Canvas1*", come dovrebbe essere in maniera predefinita.

Fatta l'interfaccia, muoviamoci sul Blocks Editor per "programmare" la logica necessaria all'applicazione. Prevediamo cinque differenti eventi:

1. Al clic sul bottone "*Bianco*" dobbiamo cambiare il "*PaintColor*" di "*Canvas1*", impostandolo sul bianco.

PUBBLICARE SUL MARKET

Avete realizzato un'applicazione pronta per il mondo reale? Volete distribuirla attraverso l'Android Market di Google? Connettetevi al seguente indirizzo:
http://market.android.com/publish
Attenzione però al fatto che, per iscriversi al market e pubblicare le proprie applicazioni, è necessario pagare 25 dollari con una carta di credito.

2. Al clic sul bottone "*Giallo*" dobbiamo cambiare il "*PaintColor*" di "*Canvas1*", impostandolo sul giallo.
3. Al clic sul bottone "*Rosso*" dobbiamo cambiare il "*PaintColor*" di "*Canvas1*", impostandolo sul rosso.
4. Al clic sul bottone "*Pulisci*" dobbiamo ripulire il contenuto di "*Canvas1*".
5. In qualche maniera, quando l'utente trascina il dito all'interno di "*Canvas1*", dobbiamo far tracciare una linea del colore impostato.

Vediamo, passo dopo passo, come realizzare questi cinque punti. Cominciamo dal bottone "*ButtonBianco*". Trasciniamo sul campo il gestore di evento «*when ButtonBianco.Click do*». Come conseguenza dell'azione incastriamo al suo interno il blocco «*set Canvas1.PaintColor to*». Il colore da collegare a questo blocco possiamo sceglierlo dal gruppo "*Colors*" della sezione di blocchi "*Built-in"*. Naturalmente, per "ButtonBianco", scegliete il colore "white". Fate lo stesso per "ButtonGiallo" e "ButtonRosso", usando i colori "*yellow*" e "*red*".
Al componente "*ButtonPulisci*", invece, deve essere associata un'azione differente. Il blocco da incastrare è tra quelli di "*Canvas1*", e per l'esattezza è l'azione «*call Canvas1.Clear*» (cioè «*ripulisci Canvas1*»).
Ora dobbiamo fare in modo che trascinando il dito su *Canvas1* vengano disegnate delle linee. L'evento da gestire corrisponde al blocco «*when Canvas1.Dragged do*». Trasciniamolo sull'area di lavoro del Blocks Editor. Questo blocco di gestione evento, come avrete sicuramente notato, è un po' più complesso di quelli utilizzati per gestire i clic sui bottoni. L'evento di clic, infatti, è molto semplice e non ha argomenti o varianti sul tema. Muovere il dito sullo schermo, invece, è un'azione più complessa.
In questo caso non basta sapere che il dito si è mosso: bisogna anchesapere da dove è partito e fin dove è arrivato. Proprio per riportare questo genere di informazioni l'evento «*when Canvas1.Dragged do*» aggancia automaticamente i blocchi di tipo «*name ...*». Questi blocchi realizzano ciò che, in gergo, è chiamata una variabile. Quando l'utente trascina il dito, l'evento riporterà le informazioni aggiuntive all'interno delle variabili agganciate. A noi, in particolar modo, interessa sapere che *prevX* e *prevY* riporteranno le coordinate che aveva il dito prima dell'evento, mentre *currentX* e *currentY* riporteranno quelle raggiunte al termine del movimento. Sfruttando questa informazione possiamo allora inserire la conseguenza dell'evento, trascinando al suo interno un blocco «*call Canvas1.DrawLine*».Per disegnare una linea, come spiegato prima, ci vogliono le quattro coordinate che indicano da dove la linea parte (*x1* e *y1*) e dove la linea finisce (*x2* e *y2*), ed infatti il blocco che abbiamo aggiunto dispone degli agganci necessari per queste quattro informazioni. Non dobbiamo far altro che collegare ciascuno di questi tasselli con la corrispettiva variabile. I blocchi utili per compiere questa operazione sono presenti nella voce "*My Definitions*". Trascinate nell'are di lavoro i blocchi «*value prevX*», «*value prevY*», «*value currentX*» e «*value currentY*», agganciandoli, rispettivamente, ai tasselli *x1, y1, x2* e *y2*.

Fig. 17: La logica per tracciare una linea ad ogni movimento del dito sul componente "Canvas1"

L'AUTORE

Carlo Pelliccia lavora presso 4IT (www.4it.it), dove si occupa di analisi e sviluppo di applicazioni server e mobili. Nella sua carriera di technical writer ha pubblicato cinque manuali ed oltre duecento articoli, molti dei quali proprio tra le pagine di ioProgrammo. Il suo sito, che ospita anche diversi progetti Open Source, è disponibile all'indirizzo www.sauron-software.it

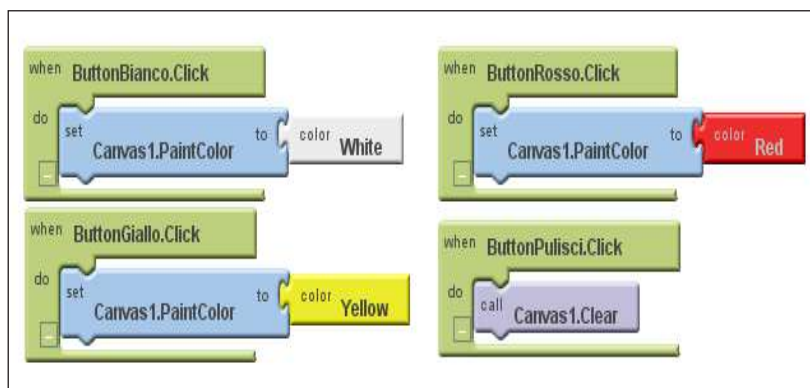



Fig. 16: La logica per la gestione dei quattro pulsanti

CONCLUSIONI

Attraverso una coppia di esempi pratici abbiamo esplorato le principali possibilità di App Inventor, La piattaforma, naturalmente, dispone di ulteriori caratteristiche che permettono di spingersi più in là nella realizzazione di applicazioni complesse ed adatte al mondo reale. Anche gli aspetti più avanzati di App Inventor, ad ogni modo, sono semplici da apprendere ed impiegare. Potete a questo punto approfondire da voi, sperimentando i tanti componenti a disposizione nella palette sulla sinistra e, se masticate un po' di inglese, seguendo alcuni tra i tanti tutorial a disposizione nella sezione "*Learn*".

Carlo Pelliccia

PORTA TWITTER SU GOOGLE ANDROID

IN QUESTO ARTICOLO VEDREMO COME SVILUPPARE UN'APPLICAZIONE PER ANDROID, CAPACE DI DIALOGARE CON IL SERVIZIO DI SOCIAL NETWORKING TWITTER. A TAL SCOPO MOSTREREMO COME UTILIZZARE LA LIBRERIA TWITTER4J

Conoscenze richieste
Java

Software
Twitter4J, Eclipse. Android SDK

Cosa stai facendo? Basta rispondere a questa semplicissima domanda per cominciare a utilizzare uno dei servizi di social network che sta esponenzialmente aumentando le registrazioni e sta prepotentemente imponendosi come una delle più importanti piattaforme in questo campo. Una delle tante definizioni possibili per twitter ce la dà Wikipedia: "*Twitter è un servizio di social network e microblogging che fornisce agli utenti una pagina personale aggiornabile tramite messaggi di testo con una lunghezza massima di 140 caratteri.*
Gli aggiornamenti possono essere effettuati tramite il sito stesso, via SMS, con programmi di messaggistica istantanea, e-mail, oppure tramite varie applicazioni basate sulle API di Twitter".
Un concetto centrale nel funzionamento di Twitter è quello del following e dei follower.
Il primo sta a indicare gli utenti che abbiamo deciso di seguire, ossia quegli utenti i cui *tweet* (leggi nota) ci verranno notificati e andranno a costituire la nostra friend timeline.
Simmetricamente a quanto appena detto, i follower costituiscono la lista di tutti quegli utenti che hanno deciso di seguirci. Naturalmente non è assolutamente obbligatorio seguire uno dei propri follower e viceversa. Uno dei punti di forza di Twitter è proprio quello di consentire una lunghezza massima di 140 caratteri per ogni post, anzi, per usare il gergo di Twitter, per ogni *Tweet*. Anche se a prima vista ciò possa apparire come una grossa limitazione rispetto ad altri strumenti come i normali blog o Facebook, grazie a ciò è possibile utilizzarlo con un gran numero di strumenti (PC, telefonini, palmari etc...) e di applicazioni (vedi definizione di Wikipedia). Inoltre questa estrema necessità di sintesi nel comunicare ha fatto sì che l'utilizzo di questo servizio sia andato al di là di quello inizialmente previsto dagli stessi ideatori. Ne faremo brevemente due esempi, ma se ne potrebbero elencare molti altri, giusto per cercare di capire in che direzione si sta muovendo la Rete.

1 – molte testate di giornali, o di news in generale, (specialmente quelle americane e inglesi) usano Twitter come mezzo per diffondere le ultime notizie. Avendo a disposizione solamente 140 caratteri compongono un Tweet di pochissime parole seguito da un link all'articolo vero e proprio. Naturalmente un URL può essere relativamente lungo, soprattutto se paragonato alla limitata disponibilità di testo di Twitter. Per questo si utilizza solitamente degli strumenti tipo *TinyURL*; questo è un ulteriore esempio di come possano nascere sinergie tra strumenti e servizi fra loro eterogenei.
2 – una cosa che mancava a Twitter era proprio la possibilità di ricercare tutti quei Tweet che vertessero sullo stesso argomento. A tale lacuna sono stati gli stessi utenti a porre rimedio; infatti si è utilizzato il vecchio sistema dei tag. Quando si scrive qualcosa che si reputa essere di interesse generale, lo si termina con un tag del tipo *#topic*, in questo modo, chiunque voglia rispondere o dire qualcosa sull'argomento, potrà farlo terminando a sua volta il Tweet con il medesimo tag. Un esempio celebre di questo tipo di utilizzo lo si è avuto a seguito delle elezioni in Iran, dove grazie a Twitter e al tag *#iranelection* i manifestanti riuscivano in qualche modo a tenere informata l'opinione pubblica su ciò che stava accadendo nel proprio paese.

Adesso che abbiamo dato una brevissima introduzione su cosa sia Twitter veniamo alla parte maggiormente vicina ai nostri interessi, cioè lo sviluppo di applicazioni.

API PER I WEB SERVICE DI TWITTER

In un precedente articolo abbiamo già parlato dei *Web Service REST-style*, in particolare di cosa siano, a cosa servano e come vadano utilizzati.

Fortunatamente, da qui a poco, andremo a utilizzare delle API per Java che ci "nascondono" in gran parte i meccanismi di comunicazione che il web service di Twitter richiede. È però sempre bene sapere cosa stia "succedendo" all'interno della libreria che utilizzeremo, soprattutto in visione del refactoring che sarà necessario effettuare sulla libreria stessa per realizzare il porting sulla piattaforma Android.
REST è un acronimo che sta per "*Representational State Transfer*" e, cosa ancora più importante, non è uno standard, bensì un An Architectural Style.
Senza addentrarci in discussioni più filosofiche che di ingegneria del software facciamo subito un esempio mirato:

Formats:
xml, json, rss, atom

HTTP Method(s):
GET

Requires Authentication:
true

API rate limited:
1 call per request

Parameters:

- since_id. Optional. Returns only statuses with an ID greater than (that is, more recent than) the specified ID
  - Example: http://twitter.com/statuses/friends_timeline.xml?since_id=12345
- max_id. Optional. Returns only statuses with an ID less than (that is, older than) or equal to the specified ID
  - Example: http://twitter.com/statuses/friends_timeline.xml?max_id=54321
- count. Optional. Specifies the number of statuses to retrieve. May not be greater than 200
  - Example: http://twitter.com/statuses/friends_timeline.xml?count=5
- page. Optional. Specifies the page of results to retrieve.
  - Example: http://twitter.com/statuses/friends_timeline.rss?page=3

Fig. 1: Formato di una request per ottenere la friends-timeline dello user specificato nella fase di autenticazione

in Fig. 1 sono riportati i parametri necessari per ottenere la friends-timeline dello user specificato nella fase di autenticazione. Tralasciando per ora la parte che concerne l'autenticazione, vediamo come funziona la comunicazione tra un generico client e il web service.
In base a quanto riportato in Fig.1 (che non è altro che la documentazione delle REST API di Twitter consultabile all'indirizzo http://apiwiki.twitter.com/Twitter-REST-API-Method%3A-statuses-friends_timeline) per ottenere gli ultimi 25 tweet della nostra friends-timeline codificati in formato XML sarà necessario comporre un URL del tipo http://twitter.com/statuses/friendstimeline.xml?count=25
A questo punto il web service risponderà come segue:

```
<?xml version="1.0" encoding="UTF-8"?>
<statuses>
    <status>
<created_at>Tue Apr 07 22:52:51 +0000 2009</created_at>
<id>1472669360</id>
<text>At least I can get your humor through tweets. RT @abdur: I don't mean this in a bad way, but genetically speaking your a cul-de-sac.</text>
<source><a href="http://www.tweetdeck.com/">TweetDeck</a></source>
<truncated>false</truncated>
<in_reply_to_status_id></in_reply_to_status_id>

<in_reply_to_user_id></in_reply_to_user_id>
<favorited>false</favorited>
<in_reply_to_screen_name></in_reply_to_screen_name>
<user>
<id>1401881</id>
 <name>Doug Williams</name>
 <screen_name>dougw</screen_name>
 <location>San Francisco, CA</location>
 <description>Twitter API Support. Internet, greed, users, dougw and opportunities are my passions.</description>
<profile_image_url>http://s3.amazonaws.com/twitter_production/profile_images/59648642/avatar_normal.png</profile_image_url>
 <url>http://www.igudo.com</url>
 <protected>false</protected>
 <followers_count>1027</followers_count>
<profile_background_color>9ae4e8</profile_background_color>
 <profile_text_color>000000</profile_text_color>
 <profile_link_color>0000ff</profile_link_color>
<profile_sidebar_fill_color>e0ff92</profile_sidebar_fill_color>
<profile_sidebar_border_color>87bc44</profile_sidebar_border_color>
 <friends_count>293</friends_count>
 <created_at>Sun Mar 18 06:42:26 +0000 2007</created_at>
 <favourites_count>0</favourites_count>
 <utc_offset>-18000</utc_offset>
 <time_zone>Eastern Time (US & Canada)</time_zone>

<profile_background_image_url>http://s3.amazonaws.com/twitter_production/profile_background_images/2752608/twitter_bg_grass.jpg</profile_background_image_url>

<profile_background_tile>false</profile_background_tile>
 <statuses_count>3390</statuses_count>
 <notifications>false</notifications>
 <following>false</following>
 <verified>true</verified>
</user>
     </status>
     ... truncated ...
</statuses>
```

Per motivi di spazio, non riportiamo tutta la risposta per intero. Ad ogni modo i tag *status* all'interno di *statuses* si ripetono con la medesima struttura.

CURIOSITÀ
Il nome "*Twitter*", corrispondente sonoro della parola *tweeter*, deriva dal verbo inglese *to tweet* che significa "cinguettare". Tweet è anche il termine tecnico degli aggiornamenti del servizio

Ora, se dovessimo fare tutto da zero dovremmo fare delle request via http, aprire un socket per ottenere lo stream in lettura; una volta ottenuto andrebbe letto e parsato a seconda del formato specificato (*xml, json, rss, atom*) al momento della request.
Fortunatamente esistono diverse librerie per Java che si fanno già carico di tutto questo lavoro e ci mettono a disposizione già delle classi ben formate. Abbiamo comunque ritenuto significativo riportare un esempio di risposta in XML, perché le gerarchie e la composizione delle classi ricalca proprio la struttura dello XML stesso, perciò avere un'idea di come esso sia strutturato significa riuscire già a utilizzare la libreria in maniera corretta.

NOTA

TINYURL

TinyURL è un sito web che diminuisce la lunghezza di un URL per poterla comunicare facilmente via email o Instant Messaging

LA LIBRERIA TWITTER4J

La libreria *Twitter4j* è reperibile all'indirizzo http://yusuke.homeip.net/twitter4j/en/index.html ed è una delle API Java per Twitter maggiormente diffuse. Nel sito citato poc'anzi trovate elencate tutte le caratteristiche di tale libreria. In particolare a noi interessa che sia scritta al 100% in Java e che sia compatibile con Android. In realtà questa ultima feature non è completamente esatta, ma ci occuperemo di ciò a tempo debito.
Cominciamo, invece, a scrivere un po' di codice. Sia per la stesura di questo articolo, sia del prossimo, è stato utilizzato come ambiente di sviluppo Eclipse. A ogni modo, chiunque abbia un proprio IDE preferito diverso da Eclipse, può comunque applicare delle procedure simili a quelle che andremo a descrivere.
Scaricate lo zip che trovate sul sito sopra citato (la versione 2.0.8 è l'ultima al momento della stesura dell'articolo). Create poi un nuovo progetto che io ho chiamato *Twitter*. Per adesso non siamo interessati ai sorgenti, perciò creiamo una sotto-directory sulla root del progetto che chiamiamo *lib* e copiamo poi tutti i jar contenuti nella directory *lib* dello zip nella nostra directory appena creata. Copiamo inoltre anche il file *twitter4j-2.0.8.jar* sempre nella stessa directory. A questo punto andiamo sulle proprietà del progetto (cliccando con il tasto destro sul progetto e selezionando *Properties...*) e selezioniamo la voce *Java build path* e poi il tab *Libraries*. Premendo il pulsante *Add Jars...* selezionate tutti i jar che abbiamo precedentemente copiato nella directory *lib*. Una volta fatto tutto ciò, vi dovreste trovare in una situazione analoga a quella riportata in Fig. 2. È quindi tutto pronto per effettuare i primi test con la libreria Twitter4j. Vediamo innanzitutto come ottenere gli ultimi 25 tweet, così come nell'esempio precedente:

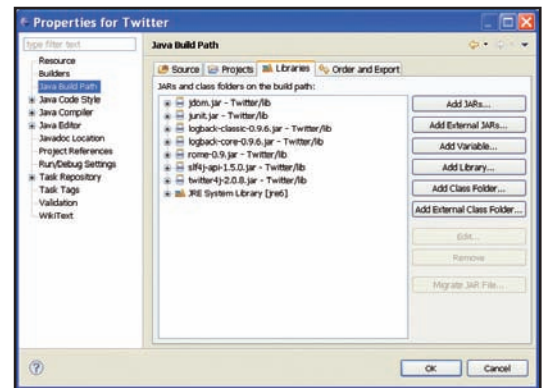



Fig. 2: Insieme delle librerie necessarie al progetto Twitter

```
public static void main(String[] args) throws
                                        TwitterException
{
 Twitter twitter = new
                    Twitter("userName","password");
 Paging pag = new Paging();
 pag.setCount(25);
 List<Status> statusList =
                    twitter.getFriendsTimeline(pag);
 for (Status status : statusList)
 {
     System.out.println(status.getId() +" - "+
                                   status.getText());
 }
}
```

Per prima cosa è stato istanziato un oggetto della classe *Twitter* al cui costruttore abbiamo passato il nostro user-name e la nostra password.
Quando viene creato tale oggetto, si stabilisce già una connessione verso il web service e si occupa anche dell'autenticazione.
Subito dopo abbiamo creato un oggetto *Paging*; possiamo considerare tale classe in maniera equivalente a un cursore nei database. I metodi messi a disposizione da tale classe sono diversi; per adesso ci possiamo accontentare di impostare semplicemente il numero di tweet che vogliamo ci vengano restituiti.
L'invocazione del metodo *getFriendsTimeline*, che prende come argomento proprio l'oggetto *paging*, restituisce una lista di oggetti *status*.
Avrete certamente notato come la struttura delle classi ricalchi perfettamente quella dello XML precedentemente riportato: infatti, dentro il tag *<statuses>* (che corrisponde alla nostra lista di oggetti *status*) si susseguono una serie di tag *<status>* figli. A sua volta tale tag ha un serie diversificata di sotto-figli, per ognuno di essi la classe *Status* mette a disposizione dei relativi metodi *getters* che restituiscono un dato già tipizzato. Nel nostro esempio abbiamo deciso di

stampare l'id, lo user e il testo di ogni status della FriendsTimeline. Con il mio account ho ottenenuto il seguente output sulla console:

```
2691267627 - TIME.com - For #ff, add @time_live. Starting Monday, it's the best way for you to communicate 1-on-1 with our newsroom. And we'll follow you back!
2691154798 - developerworks - Learn how to create drag-and-drop code that is modular, and easier to write and maintain > http://su.pr/6bXlqP
2690818806 - Andrea Galeazzi - I'm watching rock & roll circus...amazing!
2690608685 - TIME.com - Why girls have BFFs and boys hang out in packs | http://su.pr/92EuL5
2690262831 - developerworks - Parallel development for mere mortals - Step through Subversion version-control > http://su.pr/2leWWm
2689450274 - developerworks - Writing great code with FileNet P8 APIs - Reliable-Scalable-Highly-Available content management > http://su.pr/19PsCq
…….
```

Vediamo invece come inviare i nostri tweet utilizzando la libreria Twitter4j: in questo caso la procedura è anche più semplice di quella vista poco fa. Infatti, una volta istanziato l'oggetto twitter che si occupa della connessione e dell'autenticazione, basta invocare il metodo *updateStatus* come segue:

```
Status status = twitter.updateStatus("That's an example about how to tweet by Twitter4j");
System.out.println("Successfully updated the status to [" + status.getText() + "].");
```

Tale codice produce il seguente output:

```
Successfully updated the status to [That's an example about how to tweet by Twitter4j]
```

ANDROID E IL SUO EMULATORE

Potremmo continuare a esporre ulteriori funzionalità della libreria Twitter4j, ma non dobbiamo dimenticare che il nostro scopo è quello di implementare un client di Twitter che giri sulla piattaforma Android. È quindi opportuno, a questo punto, iniziare a introdurre i concetti base su cui si fonda il sistema operativo di Google.
Android è una piattaforma per dispositivi mobili che include sistema operativo, middleware e applicazioni di base. Maggiori informazioni, comunque, sono reperibili nel precedenti articolo di questa cover story). Per essere quanto più pratici possibile, vediamo subito di mettere in piedi un ambiente che ci permetta di sviluppare e debugare la nostra applicazione. Dopo aver installato e scaricato Android, una delle parti più importanti è quella della creazione del Virtual Device. Infatti è proprio grazie a questo punto che saremo in grado di emulare un generico dispositivo sul nostro PC. Per fare ciò basta lanciare da prompt il comando Android *create avd --target 2 --name my_avd*. La prima parte del comando (*create avd*) indica che vogliamo creare un nuovo *Android Virtual Device*, la seconda (*--target 2*) fa sì che tale device possa girare sull'emulatore, mentre la terza (*--name my_avd*) imposta semplicemente il nome del device che andremo a creare. A questo punto, una volta lanciato tale comando, vi verrà chiesto se volete creare un profilo hardware personalizzato; noi possiamo tranquillamente accettare quello di default e quindi premere invio. Ora il nostro device è pronto per essere fatto girare sotto l'emulatore.

L'AMBIENTE DI SVILUPPO

Quando parliamo di programmazione in Java, indipendentemente da ciò che stiamo sviluppando, difficilmente possiamo fare a meno di Eclipse ed anche questo caso non fa eccezione.
Per sviluppare sulla piattaforma Android con Eclipse è necessario installare il relativo plug-in e configurarlo opportunamente.
Utilizzando l'ultima versione di Eclipse al momento, Galileo, basta che andiate su *Help Install New Software* e clicchiate sul bottone Add. Inserite poi nel campo Location https://dl-ssl.google.com/android/eclipse/ e scegliete il nome che preferite nel campo *Name*, premete poi *OK*. Così facendo dovrebbe apparire una check box con il nome "*Developer Tools*", selezionatela e premete *Next* e poi *Finish*. Se tutto procede correttamente alla fine vi verrà chiesto di riavviare Eclipse. Una volta riavviato andate su *Window Preferences* e selezionate *Android* sulla parte destra.
Ciò che è necessario impostare è la location di dove abbiamo posizionato l'SDK di Android (nel nostro caso *D:\Program Files (x86)\android-sdk-windows-1.5_r2*).
Una volta selezionato correttamente il percorso dovreste ritrovarvi in una situazione del tutto simile a quella riportata in Fig. 4. Anche se quest'ultima parte può risultare tediosa è fondamentale che venga correttamente eseguita in modo tale da evitare che durante la fase di sviluppo qualcosa non funzioni senza capirne il motivo.

SMARTPHONE ANDROID

HTC è stata la prima azienda al mondo a mettere in vendita uno smartphone con installato il sistema operativo Android. Dopo aver lanciato sul mercato i vari HTC Dream, HTC Magic e l'ultimissimo HTC Hero, ora l'azienda di Taiwan vorrebbe arrivare sul mercato con un nuovo smartphone Android: l'HTC Click.

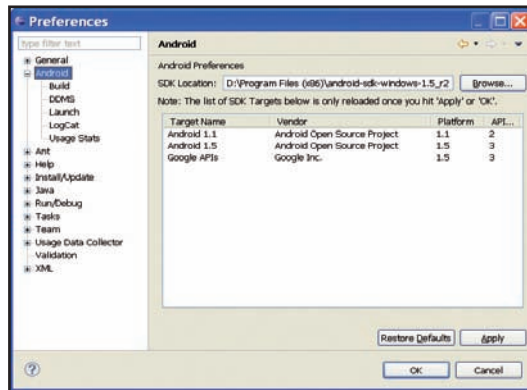



Fig. 4: Configurazione del plug-in per Android

ANDROID, HELLO WORD!

Dopo tanta fatica possiamo finalmente far girare la nostra prima applicazione su Android e, come la "tradizione" ormai impone, anche noi non potevamo esimerci dal cominciare proprio con un bel Hello Word! Per prima cosa andate su *File New Other.Android Project* e compilate i seguenti campi:

Project name: Hello Word
Application name: Hello, word
Package name: it.ioprogrammo
Create *Activity*: HelloWord
Min SDK *Version*: 3

Dentro il package *it.ioprogrammo* troverete così la classe *HelloWord* che deriva dalla classe *Activity*, che è presente nel framework di Andriod e di cui parleremo più diffusamente nel prossimo articolo. Per adesso limitiamoci a constatare che tale classe sovrascrive il metodo *onCreate* come segue:

```
public class HelloWord extends Activity { @Override
    public void onCreate(Bundle savedInstanceState){
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main); }
}
```

Tale metodo viene invocato non appena questa attività viene creata. Purtroppo, in questo articolo, non abbiamo sufficiente spazio per approfondire cosa si intenda per attività né per parlare della creazione dei layout, limitiamoci quindi a scrivere il codice necessario per visualizzare la scritta Hello Word sul display del nostro Virtual Device:

```
package it.ioprogrammo;
import android.app.Activity;
import android.os.Bundle;
import android.widget.TextView;
public class HelloWord extends Activity {
@Override
public void onCreate(Bundle savedInstanceState) {
   super.onCreate(savedInstanceState);
   TextView tv = new TextView(this);
   tv.setText("Hello, Android");
   setContentView(tv); }
}
```

Non abbiamo fatto altro che creare un oggetto TextView che è in grado di visualizzare un testo e poi l'abbiamo "appiccicato" al display mediante il metodo *setContentView(tv);* non ci resta che avviare l'activity, per farlo, andate su *Run Configurations*... selezionate la voce *Android Application* e cliccate sul tasto *New.*
Chiamiamo anche questa configurazione *HelloWord* e selezioniamo il progetto *HelloWord* premendo il tasto *Browse*. A questo punto premete *Run* e finalmente dovreste ottenere quanto riportato in Fig. 4. Naturalmente d'ora in poi possiamo avviare tale configurazione semplicemente premendo il tasto *Run*.

Fig. 4: Activity Hello Word in azione sul nostro emulatore Android

L'AUTORE

Laureato in ingegneria elettronica presso l'università Politecnica delle Marche, lavora presso il reparto R&D della Korg S.p.A. Nei limiti della disponibilità di tempo risponde all'indirizzo andrea.galeazzi@gmail.com

CONCLUSIONI

In questo articolo abbiamo presentato Twitter e utilizzato le API Java Twitter4j per usufruire dei servizi messi a disposizione. Nella seconda parte abbiamo invece iniziato a presentare il nuovo OS di Google. Nel prossimo metteremo assieme questi due aspetti per realizzare il nostro client Twitter su Android!

Andrea Galeazzi

UN CLIENT TWITTER SU ANDROID

PARTE 2

CONTINUIAMO E COMPLETIAMO IL NOSTRO PROGETTO PER IMPLEMENTARE UN CLIENT TWITTER SULLA PIATTAFORMA ANDROID. L'OCCASIONE CI PERMETTERÀ DI APPROFONDIRE MOLTI ASPETTI SUL FUNZIONAMENTO DEL SISTEMA OPERATIVO CREATO DA GOOGLE

REQUISITI

Conoscenze richieste
Java

Software
Eclipse, Android SDK 1.5

Impegno

Tempo di realizzazione

Chi lavora nel campo del software ormai lo sa benissimo: le cose cambiano con una velocità elevatissima, magari un framework o una piattaforma che avevamo imparato a conoscere e utilizzare meno di un anno fa, è già parte della storia dell'informatica.
In questo articolo proveremo a mettere assieme due grosse novità di quest'anno (benché entrambe hanno avuto origine uno o due anni prima): Twitter e Android.
Visto che nello scorso articolo abbiamo dedicato maggior spazio al funzionamento di Twitter, sia come social network che come set di API, questa volta ci occuperemo prima di Android descrivendo i suoi meccanismi di funzionamento e come implementare un'applicazione che si colleghi a Internet e sia interagibile per mezzo di un touchscreen.

CREARE APPLICAZIONI PER ANDROID

Abbiamo già dedicato una parte dello scorso articolo alla discussione dell'installazione di un environment che ci permetta di sviluppare delle applicazioni in Java ed emularle su un virtual device di Android. Si rimanda quindi al precedente numero di ioProgrammo per la messa in piedi di tale ambiente; in questa sede ci limitiamo a ricordare che è necessario scaricare e installare lo SDK di Android, reperibile all'indirizzo http://developer.android.com/sdk/1.5r3/index.html ed è fortemente consigliato utilizzare anche il relativo plug-in per Eclipse. Proprio al termine di quest'ultimo avevamo iniziato a scrivere la nostra prima applicazione *Hello World*. Ripartiamo quindi da dove ci siamo lasciati la volta scorsa. Benché il codice sia abbastanza semplice già contiene vari aspetti che meritano di essere approfonditi:

```
public class HelloWord extends Activity {
    /** Called when the activity is first created. */
    @Override
```

```
public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    TextView tv = new TextView(this);
    tv.setText("Hello, Android");
    setContentView(tv);
}
}
```

La prima cosa importante da notare è che la nostra classe eredita dalla classe *Activity*. Possiamo pensare alla classe *Activity* (o meglio alla classe che la estende) come una classe che funge da "entry point" per l'applicazione che vogliamo realizzare.
A mio avviso, quando si sviluppa un'applicazione per Android, è molto semplice adottare questo tipo di visione: Android è un host all'interno del quale possono girare più Activity. Di conseguenza un'activity verrà "pilotata" da Android che gli comunicherà quando viene avviata, messa in pausa, riavviata o distrutta. La Fig. 1 illustra i vari stati in cui un'activity può venirsi a trovare e anche in che

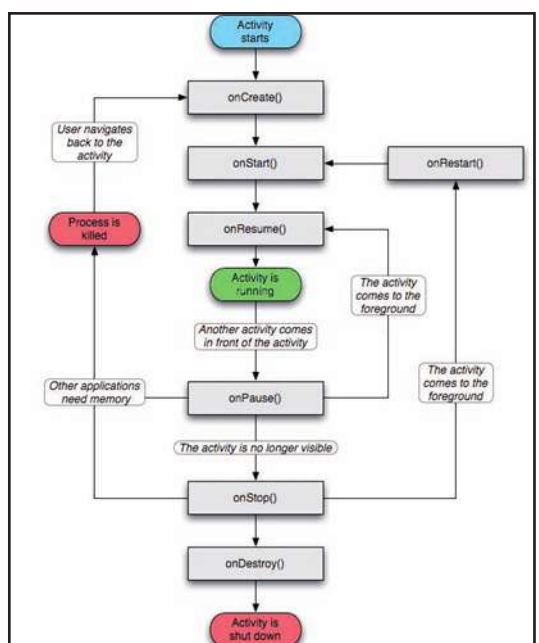



Fig. 1: State Diagramm che illustra il Life Cycle di un'activity

modo può arrivarci. Lo state diagramm in questione è già abbastanza esplicativo di per sé, vale comunque la pena spendere alcune parole al riguardo. Come potete notare, appena viene creata un'activity su di essa, vengono anche invocati nell'ordine i seguenti metodi: *onCreate*, *onStart* e *onResume*. Al termine dell'invocazione di quest'ultimo metodo possiamo dire che l'applicazione sta girando. In particolare, nel metodo *onCreate*, vengono tipicamente creati quei componenti grafici che andranno a costituire l'interfaccia grafica. Tale interfaccia viene poi infatti settata mediante il metodo *setContentView*. Quando e se un'activity viene sospesa, solitamente perché ne viene avviata un'altra, è invocato il metodo onPause. In questo punto è opportuno salvare, qualora ce ne fosse bisogno, tutti i dati che costituiscono lo stato in cui si trova l'activity.
Una volta che quest'ultima è posta nello stato *paused*, abbiamo tre possibili stati successivi in cui sarà consentito andare:*Destroyed, Stopped, Running*.
Sul primo stato c'è abbastanza poco da dire, l'applicazione viene killata (ad esempio a seguito di una chiusura), e qui è necessario rilasciare tutte le risorse eventualmente allocate all'inizio, più precisante nel metodo *onDestroy*. I restanti due casi sono invece quelli più interessanti.
Cerchiamo di comprendere come mai siamo obbligati a implementare un'applicazione che sia in grado di gestire il proprio stato corrente mediante la ricezione delle callback da parte del sistema operativo, che ci informa su quale sarà il prossimo stato nel quale la nostra activity andrà a trovarsi. Dobbiamo infatti tenere a mente che Android è stato progettato per girare su piattaforme hardware limitate e ciò comporta un'oculata gestione delle sue risorse, specialmente la memoria. Per questo motivo Android gestisce le varie Activity come uno stack. In testa si trova quella attualmente in primo piano sul display; immediatamente dopo si troverà la penultima Activity, quella che come già accennato prima, è stata messa in stato *Paused*. Tutte le altre sotto quest'ultima vengono messe nello stato stopped dopo la chiamata ad onStopped. Se poi l'activity viene richiamata in primo piano, a seconda della memoria richiesta nei momenti precedenti, possiamo ritrovarci in due situazioni differenti: o la memoria a disposizione era sufficiente e quindi verrà invocato il metodo onRestart, che alla fine ci riporterà nello stato di Running, oppure è stato necessario uccidere l'Activity e quindi ricominciare tutto ripartendo da *onCreate*.

DEFINIRE UNA GUI SU ANDROID

Sicuramente non è possibile tentare di scrivere un'applicazione per Android senza aver conoscenza della gestione del life-cycle di un'activity.
Un altro "mattoncino fondamentale" è quello dello sviluppo dell'interfaccia grafica. Nell'esempio precedente abbiamo semplicemente creato una textView tramite una new, scritto su di essa il testo *"Hello, Android"* e posta la textView sul display.
Potete facilmente intuire che se il layuot si comincia a complicare un po' diventerebbe veramente difficoltoso creare delle GUI questo modo.
Esiste però anche un altro modo per definire il layuot di un'applicazione, ed è tramite un file XML; cominciamo quindi subito con l'analizzare come è strutturato quello che definisce il layuot della nostra applicazione:

```
<?xml version="1.0" encoding="utf-8"?>
<AbsoluteLayout android:id="@+id/widget38"
                android:layout_width="fill_parent"
        android:layout_height="fill_parent" xmlns:android=
          "http://schemas.android.com/apk/res/android">
<ListView android:id="@+id/list"
android:layout_width="319px" android:layout_height="356px"
        android:layout_x="1px"android:layout_y="0px">
</ListView>
<EditText
 android:id="@+id/text" android:layout_width="259px"
android:layout_height="50px"android:background="#ffffffff"
 android:textSize="12sp" android:textColor="#ff666666"
    android:layout_x="1px" android:layout_y="383px">
</EditText>
<Button android:id="@+id/button"
              android:layout_width="wrap_content"
      android:layout_height="52px" android:text="Send"
   android:layout_x="260px" android:layout_y="383px">
</Button>
</AbsoluteLayout>
```

Un client per Twitter è concettualmente molto semplice, deve solamente assolvere a due compiti: mostrarci la nostra friends-timeline e permetterci di inviare i nostri Tweet. Per fare ciò abbiamo quindi bisogno di solo tre componenti:

1. Una ListView in cui visualizzare i vari Tweet della friends-timeline;
2. Una TextView in cui inserire ciò che vogliamo "tweetare";
3. Un bottone per inviare effettivamente quanto scritto nel controllo sopra citato.

In Fig. 2 potete vedere come si presenta graficamente l'XML appena riportato.
Notiamo che come layout di root abbiamo scelto un AbsoluteLayout all'interno del quale abbiamo posto i nostri controlli; gli attributi associati a quest'ultimi sono talmente palesi che approfondire il loro significato non sarebbe di alcuna utilità.

TOOL PER DISEGNARE L'INTERFACCIA GRAFICA

In realtà scrivere il file XML per la definizione della GUI dall'inizio alla fine può risultare un compito noioso ed alcune volte anche poco praticabile. Esiste a tal proposito un tool RAD di nome AnDroidDraw (http://www.droiddraw.org/androiddraw.html) che permette di definire un'interfaccia grafica in maniera visuale.

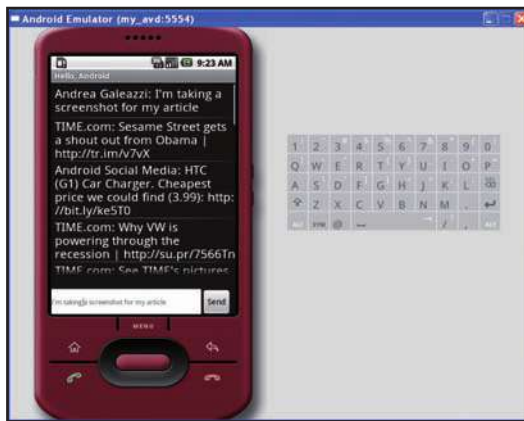



Fig. 2: Layuot dell'applicazione TwitterClient

Ciò che invece è importante iniziare a notare è come settare il layout descritto nel file XML. Veniamo quindi immediatamente a codice che setta il layout sulla *onCreate*.

```
public class TwitterClient extends Activity implements
                                        OnClickListener {
 private EditText text = null;
 private Button btn = null;
 private TwitterAdapter adapter = null;
 private Twitter twitter = null;
 @Override
 public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.main);
    ...
```

Per adesso tralasciamo il fatto che la nostra applicazione *TwitterClient* implementi anche l'interfaccia *OnClickListener* (ce ne occuperemo di qui a poco). Ciò che adesso ci interessa analizzare è la chiamata al metodio *setContentView*. Quando create un nuovo progetto Android il plug-in di Eclipse (vedi articolo precedente) definisce già una struttura predefinita che, ad esempio, affianca alla directory *src*, dove sono contenuti i sorgenti, un'altra chiamata *res* contenente a sua volta altre tre sotto directory: *drawable, layout* e *strings*.
Senza addentrarci nei dettagli di ognuna di esse, concentriamoci ad esempio sulla directory *layout* dato che è quella che ci riguarda più da vicino.
Il file *main.xml* in essa contenuto è proprio quello riportato sopra. Il plug-in di Eclipse genera automaticamente una costante numerica per ogni entità contenuta nella directory *res*:

```
/* AUTO-GENERATED FILE.  DO NOT MODIFY.
package it.galeazzi.andrea;
public final class R {
    public static final class attr {}
    public static final class drawable {
        public static final int icon=0x7f020000;
        public static final int refresh=0x7f020001;
    }
    public static final class id {
        public static final int button=0x7f050003;
        public static final int list=0x7f050001;
        public static final int text=0x7f050002;
        public static final int widget38=0x7f050000;
    }
    public static final class layout {
        public static final int main=0x7f030000;
    }
    public static final class string {
        public static final int app_name=0x7f040001;
        public static final int hello=0x7f040000;
    }
}
```

Tralasciando di descrivere nel dettaglio meccanismi interni, sia per motivi di spazio sia perché alla fine tutto risulta trasparente dal lato dello sviluppatore, possiamo affermare che è possibile recuperare la maggior parte delle risorse proprio attraverso tale file. Poco fa abbiamo visto come impostare il layout recuperando il suo *id* proprio dalla classe *R*.
Avrete di certo notato che, ad esempio, oltre alla sotto classe *layout*, ne esiste un'altra chiamata *id*, i cui membri hanno come nome proprio gli id specificati nel relativo attributo del file *main.xml*.
Ciò ci dà lo spunto per avanzare ulteriormente nell'analisi del nostro codice. Rivediamo quindi per intero il metodo *onCreate* che avevamo precedentemente troncato:

```
@Override
 public void onCreate(Bundle savedInstanceState) {
   super.onCreate(savedInstanceState);
   setContentView(R.layout.main);
   ListView lv = (ListView)findViewById(R.id.list);
   text = (EditText)findViewById(R.id.text);
   btn =(Button)findViewById(R.id.button);
   btn.setOnClickListener(this);
   try {
     twitter = new Twitter("name","password");
     adapter = new TwitterAdapter
         (twitter,this,android.R.layout.simple_list_item_1);
     lv.setAdapter(adapter);
   } catch (TwitterException e) {}
}
```

Si ha ora la necessità di recuperare i componenti definiti nell'XML come classi Java. A differenza del primo esempio, *Hello Android*, in cui facevamo esplicitamente delle *new* per i componenti che ci servivano, in questo caso dobbiamo utilizzare il metodo *findViewById*, che restituisce il componente a partire da un dato *id*, che, come al solito, ci viene fornito dalla classe *R*. Visto poi che tale metodo restituisce una generica View (classe base), è

NOTA

TWITTER4J

La libreria *Twitter4j* è reperibile all'indirizzo http://yusuke.homeip.net/twitter4j/en/index.html (è presente anche nel supporto CD-ROM che accompagna la rivista) ed è una delle API Java per Twitter maggiormente diffuse. Nel sito citato poc'anzi trovate elencate tutte le caratteristiche di tale libreria. In particolare a noi interessa che sia scritta al 100% in Java e che sia compatibile con Android

necessario effettuare un down-cast per ottenere effettivamente la classe specializzata voluta. Fate quindi attenzione a far coincidere il tipo indicato dall'id con il down-cast che effettuate sul valore di ritorno del metodo *findViewById*.

UN ADAPTER DI TWITTER PER LA LISTVIEW

Il pattern MVC è molto diffuso nelle librerie Java, e anche Android non fa eccezione in questo senso. In particolare una *ListView* va associata a un *ListAdapter* che sarà in grado di fornirle i dati da visualizzare. Tornando nuovamente ai nostri scopi, i dati, che nello specifico sono gli ultimi tweet della nostra friends-timeline, debbono essere recuperati tramite le API messe a disposizione dal web service di Twitter a cui ci collegheremo grazie alla libreria Twitter4j presentata nell'articolo precedente:

```
public class TwitterAdapter implements ListAdapter {
 private Twitter twitter;
 private ArrayAdapter<String> adapter;
 public TwitterAdapter(Twitter twitter,Context
             context, int viewId) throws TwitterException {
  super();
  this.twitter = twitter;
  adapter = new ArrayAdapter<String>(context, viewId);
  refresh();
}

public boolean hasStableIds() {
  return adapter.hasStableIds();
}
public boolean areAllItemsEnabled() {
  return adapter.areAllItemsEnabled();
}
public int getCount() {
  return adapter.getCount();
}
public String getItem(int position) {
 return adapter.getItem(position);
}
public long getItemId(int position) {
 return adapter.getItemId(position);
}
public int getItemViewType(int position) {
 return adapter.getItemViewType(position);
}
public View getView(int position, View
                      convertView, ViewGroup parent) {
 return adapter.getView(position, convertView, parent);
}
public int getViewTypeCount() {
 return adapter.getViewTypeCount();
}
public boolean isEmpty() {
 return adapter.isEmpty();
}
public boolean isEnabled(int position) {
 return adapter.isEnabled(position);
}
public void
  registerDataSetObserver(DataSetObserver observer) {
adapter.registerDataSetObserver(observer);
}
public void unregisterDataSetObserver
                          (DataSetObserver observer) {
adapter.unregisterDataSetObserver(observer);
}
public void refresh() throws TwitterException {
  adapter.clear();
  for (Status status : twitter.getFriendsTimeline()) {
                adapter.add(status.getUser().getName() + ":
                                        "+status.getText());
}
}
```

Al costruttore tale classe riceve tre parametri: il primo è l'oggetto *twitter* che ci permette di collegarci al web service, mentre gli altri due vengono passati direttamente all'oggetto *adapter*. Inoltre, è stato aggiunto il metodo *refresh*, non previsto dall'interfaccia *ListAdapter*. È forse questo il metodo più interessante, poiché è colui il quale si occupa di riempire l'oggetto *adapter* con i tweet che recuperiamo tramite la libreria Twitter4j (più precisamente per mezzo dell'oggetto *twitter* passato al costruttore). Si è infatti scelto di preferire l'incapsulamento all'ereditarietà. Per essere più chiari abbiamo fatto questo: dovevamo implementare un'interfaccia (*ListAdapter*) ed esiste una classe (*Array Adapter*) che, oltre a fornire una implementazione per tale interfaccia, ha un comportamento molto simile a quello che vogliamo ottenere. Abbiamo così dichiarato un membro privato (ossia abbiamo incapsulato) di tale classe e abbiamo definito una serie di metodi delegate che rigirano la chiamata sulla classe incapsulata, che perciò svolge effettivamente il lavoro per conto della classe contenitrice. In questo modo, l'unico metodo suppletivo che abbiamo avuto veramente la necessità di implementare, è stato, come già accennato, il metodo *refresh*.

CATTURARE GLI EVENTI DEL DISPLAY

Rifacendoci al pattern MVC, possiamo affermare che fin qui abbiamo discusso e analizzato le parti View (ossia i componenti grafici) e Model (vale a dire *TwitterAdapter*); non ci rimane, quindi, che occuparci del Controller.
In precedenza è già stato anticipato che avremmo

PATTERN MVC

È un pattern architetturale molto diffuso nello sviluppo di interfacce grafiche di sistemi software object-oriented. Originariamente impiegato dal linguaggio Smalltalk, il pattern è stato esplicitamente o implicitamente integrato da numerose tecnologie moderne, come framework basati su PHP, su Ruby (Ruby on Rails), su Python (Django), su Java (Swing, JSF e Struts), su Objective C e su .NET.

avuto la necessità di recuperare dei componenti definiti nell'XML. Uno dei motivi è quello di riuscire a intercettare l'evento di pressione del bottone *send*; per ottenere ciò si è proceduto come segue:

```
btn =(Button)findViewById(R.id.button);
btn.setOnClickListener(this);
…..
@Override
public void onClick(View v)
{
  try
  {
     String txt = text.getText().toString();
     twitter.updateStatus(txt);
     adapter.refresh();
} catch (TwitterException e)
{
text.setText(e.getMessage());
}
}
```

Niente di più semplice: la nostra Actvity implementa anche l'interfaccia *OnClickListener* e ne implementa perciò il metodo *onClick*. La logica in esso contenuta è estremamente lineare:

1. si recupera il testo da inviare dalla TextView;
2. si spedisce tale testo al web service tramite la libreria *Twitter4j;*
3. infine si aggiorna l'adapter poiché è sicuramente entrato un altro tweet nella nostra time-line.

Impostata l'Activity come listener del bottone, ogni volta che quest'ultimo verrà premuto, sarà invocato il metodo appena descritto.

IMPORTARE LA LIBRERIA TWITTER4J SU ANDROID

Fin qui abbiamo sempre dato per scontato che nel nostro progetto fosse presente la libreria Twitter4j che abbiamo utilizzato basandoci sull'esperienza fatta nell'articolo precedente. È anche vero che, sempre nello scorso articolo, avevamo accennato al fatto che Android non implementa una Java VM classica, bensì la Dalvik virtual machine. Molto sinteticamente diciamo che essa è ottimizzata per sfruttare la poca memoria presente nei dispositivi mobili, consente di far girare diverse istanze della macchina virtuale contemporaneamente e nasconde al sistema operativo sottostante la gestione della memoria e dei thread. Ciò che interessa noi come sviluppatori è il fatto che il bytecode con cui lavora non è Java. Ne consegue che non è possibile referenziare direttamente i jar tipicamente distribuiti, ma è necessario ricompilare i sorgenti. Essendo Twitter4j una libreria open source, possiamo tranquillamente prelevare i sorgenti ed inserirli come package nel nostro progetto. Sfortunatamente ciò non è ancora sufficiente. Infatti, in alcune limitate porzioni del codice, tale libreria fa uso delle librerie DOM per l'XML.
Per ovvie ragioni di occupazione in memoria di un oggetto DOM, Android, essendo progettato per architetture embedded, non fornisce una completa interfaccia di questo tipo per il parsing di un XML bensì solo una di tipo SAX. D'altro canto i punti dove viene utilizzata sono ben localizzati e non sono così vitali per il funzionamento complessivo della libreria stessa.
Per ragioni di spazio non riporteremo le piccole modifiche apportate alla libreria, ad ogni modo, il codice nel CD allegato contiene già la libreria modificata per essere compilata per Android.
Ci rimane un'ultima cosa da fare affinché la nostra applicazione sia completamente funzionante su Android: concedere i permessi di connessione a Internet.
Per fare ciò aprite il file *AndroidManifest.xml* e cliccate su *Add* e selezionate *Uses Permession*; dalla combo box selezionate poi *android.permission. INTERNET*. Dovreste così ritrovarvi in una soluzione analoga a quella riportata in Fig. 3.

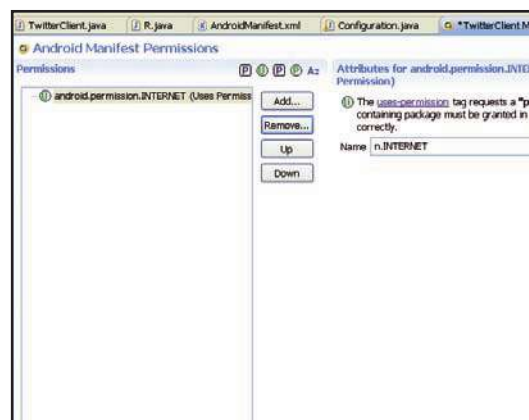



***Fig. 3:** Editor di Android Manifest in cui sono stati aggiunti i permessi di accesso a Internet*

CONCLUSIONI

Abbiamo così ultimato un client Twitter per Android, che ci ha permesso di acquisire un discreto know-how sia riguardo il sistema operativo di Google, sia riguardo la nuova piattaforma di social network che sta vedendo tassi di crescita enormi.
Naturalmente il client qui sviluppato implementa solamente le funzionalità base di Twitter. Avete ormai però tutte le conoscenze necessarie per estenderle.

Andrea Galeazzi

L'AUTORE

Laureato in ingegneria elettronica presso l'università Politecnica delle Marche, lavora presso il reparto R&D della Korg S.p.A. Nei limiti della disponibilità di tempo risponde all'indirizzo andrea.galeazzi@gmail.com

ANDROID DIALOGA CON OUTLOOK

IL PARADIGMA DEL "DATA ON THE CLOUD" RISULTA COMODO QUANDO SI VOGLIONO GESTIRE LE STESSE INFORMAZIONI DA DIVERSI CLIENT, ETEROGENEI TRA LORO. IN QUESTO ARTICOLO LO ADOPEREREMO PER TENERE SINCRONIZZATE DELLE NOTE TRA ANDROID E OUTLOOK

I cellulari di ultima generazione hanno portato la gestione dei dati in mobilità a livelli mai sperimentati fino ad oggi: si possono consultare mappe satellitari che mostrano la nostra posizione, visionare documenti e condividerli con i collaboratori, raccontare in tempo reale ciò che ci accade con foto e video. In tutti questi scenari le informazioni sono memorizzate in un server, con i client che possono leggerle, modificarle, crearne di nuove. In casi particolari i client memorizzano in locale una copia di quanto presente sul server e, grazie ad apposite procedure di sincronia, questa viene tenuta coerente con i dati originali. Lo scopo di questo articolo è quello di realizzare un sistema in grado di gestire e sincronizzare delle note tra Outlook e un dispositivo Android. Costruiremo prima di tutto un server dove memorizzare le nostre note, configureremo Outlook per accedervi ed estenderemo un'applicazione Android già esistente, aggiungendo la capacità di sincronizzare le note che questa già gestisce con il nostro server. Una volta capito il meccanismo di base, si potranno agganciare ulteriori applicazioni, come Thunderbird, un programma desktop o un sito web.

Conoscenze richieste
Java

Software
Java SDK (JDK) 5+, Eclipse 3.3+

Impegno

Tempo di realizzazione

ARCHITETTURA DELL'APPLICAZIONE

Partendo da una panoramica di massima, un'infrastruttura in grado di gestire la sincronizzazione dei dati tra diversi client deve essere composta, almeno, dai seguenti elementi:

1. Un server "on the cloud" dove memorizzare i dati.
2. Un motore di sincronia che sappia tenerli allineati tra diversi client, capace di eseguire operazioni di creazione, aggiornamento, cancellazione, risoluzione dei duplicati e notifica dei cambiamentiun protocollo con cui scambiare i dati tra i client e il motore.
3. Un client per Android e uno per Outlook che permettano di gestire e sincronizzare in locale queste informazioni, nel nostro caso le note di Outlook.

È chiaro che realizzare tutto da zero risulterebbe oltremodo oneroso, ma per fortuna tante tessere di questo mosaico esistono già. Partiamo dal protocollo: invece di inventarcene uno, con un grande dispendio di tempo, possiamo usare qualcosa di già esistente, con il vantaggio di avere alle spalle anni di utilizzo e librerie già pronte: stiamo parlando di *SyncML*. Questo protocollo nasce proprio allo scopo di gestire le operazioni di sincronizzazione dei dati tra server e smart device, conservando la massima libertà sul tipo di dato scambiato. Nelle sue specifiche, vengono definiti i tipi di sincronia possibili, i flussi di ognuna di queste, codici di errore, formato dei messaggi che si possono scambiare in ogni fase, casistiche ed esempi di utilizzo.
Anche per quanto riguarda la parte server possiamo risparmiare tempo sfruttando qualcosa di già esistente, come il software realizzato da Funambol. *Funambol Community Edition* è la versione open source di una piattaforma per la sincronia "on the cloud" di dati personali come contatti, calendario, note. Con un'approssimazione riduttiva possiamo dire che realizza i primi due punti dell'elenco precedente e utilizza, per lo scambio dei dati con i client, proprio il protocollo *SyncML*. Laddove non supportato nativamente, vengono messi a disposizione dei plug-in per effettuare la sincronia: ne esistono per la maggior parte dei sistemi operativi mobile, da Symbian ad iPhone, troviamo plugin per Outlook e Thunderbird, sono stati realizzati connettori per eGroupware del calibro di Exchange e Zimbra. Esistono SDK per Java, J2ME e C++, per aggiungere capacità di sincronia con un server Funambol all'interno di tutte le applicazioni che ancora non la supportano. Rimane scoperta solo la parte riguardante il client per Android. Su questo aspetto, terminata la creazione dell'infrastruttura, andremo a focalizzare la nostra attenzione.

Fig. 1: Una volta sincronizzati i dati con il server Funambol, è possibile accedervi anche dal web, tramite my.funambol.com

CREAZIONE DELL'INFRASTRUTTURA

Prima di tutto, occorre configurare l'ambiente di test. Come dicevamo, la logica per lo storaging delle informazioni, per il motore di sincronia e per il trasporto dei dati attraverso il protocollo SyncML 1.2.1 sono implementate dal server Funambol: abbiamo la possibilità di utilizzarne sia una versione online (http://my.funambol.com), sia di eseguire in locale la Community Edition, Basta scaricare il pacchetto per Windows o per GNU/Linux disponibile all'indirizzo http://www.forge.funambol.org/download, eseguirlo e lanciare il server tramite lo script da riga di comando:

```
bin/funambol.cmd start
```

Per il client Outlook, si può installare la versione XP o 2003 fornita con Office, oppure utilizzare la versione trial disponibile sul sito Microsoft. Occorre poi installare e configurare il *Funambol Sync Client for Microsoft Outlook*, un'estensione di Outlook che permette di sincronizzare *Contatti, Calendario* e *Note* con il server Funambol. Anch'esso si può scaricare gratuitamente dalla Funambol Forge, con la documentazione necessaria alla sua messa in opera. In ultimo, necessitiamo dei tool per creare applicazioni Android: nell'articolo faremo uso di Eclipse e dell'Android SDK, configurato secondo la guida ufficiale di Google, reperibile su http://developer.android.com/sdk.

L'ARCHITETTURA DI FUNAMBOL

La parte focale del nostro client non dovrà fare altro che richiedere al server Funanbol una sincronia delle note e aggiornare di conseguenza i propri dati, comunicando anche eventuali variazioni fatte in locale. Rispettando quanto dettato dallo standard *SyncML*, Funambol gestisce sette tipi di sincronia, di cui uno è un alert che indica al client che è necessario sincronizzarsi con il server, mentre gli altri guidano il flusso dei dati:

- *REFRESH_FROM_SERVER*: invia al client tutti gli elementi presenti sul server, sovrascrivendo completamente quanto presente sul client.
- *REFRESH_FROM_CLIENT*: invia al server tutti gli elementi presenti sul client, sovrascrivendo completamente quanto presente su server.
- *ONE_WAY_FROM_SERVER*: il server manda al client solo gli elementi nuovi, modificati o cancellati dall'ultima sincronia.
- *ONE_WAY_FROM_CLIENT*: il client manda al server solo gli elementi nuovi, modificati o cancellati dall'ultima sincronia.
- *TWO_WAY_SYNC*: il tipo di sincronia più usato, dove client e server si scambiato gli elementi nuovi, modificati o cancellati dall'ultima sincronia. Il primo a farlo è il client, poi il server.
- *SLOW_SYNC*: il client manda tutti i suoi dati al server, questo li confronta con quanto possiede, determinando quali elementi aggiungere, modificare e cancellare per allineare le due basi dati. Al termine dell'analisi, il server invia le modifiche che anche il client deve effettuare per completare l'allineamento.

Unità di base per il trasporto dei dati è il *SyncItem*, un oggetto che contiene il dato da sincronizzare e alcune informazioni a corredo. Cuore del processo di sincronia è la *SyncSource*, una classe che fa da ponte tra il server ed il client ed ha il compito di esporre i metodi per aggiungere, cancellare e modificare i dati su quest'ultimo e per sapere quali sono i nuovi elementi, quelli modificati e quelli cancellati dall'ultima sincronia, sempre sul client. Ultimo elemento del processo, il *SyncManager*, che viene chiamato per effettuare la sincronia passandogli la *SyncSource* da usare. Nel client per Outlook, ad esempio, sono state create delle *SyncSource* per i contatti, per gli appuntamenti e per le note, mentre in questo articolo ne creeremo una apposita per le note, chiamata *NoteSyncSource*.
Per capire come interagiscono questi elementi, occorre analizzare da vicino il flusso e i ruoli tipici di un'operazione di sincronia:

- Viene creata una nuova *SyncManager*, passandogli una *SyncManagerConfig* dalla quale ottenere le configurazioni per il processo di sincronia (username, password, indirizzo del server ecc).

- Viene creata una nuova *NotesSyncSource*, passandogli una *NotesSyncSourceConfig* dalla quale ottenere le configurazioni per la specifica operazione di sincronia (tipo di dati gestiti, dati remo-

SYNCML

Il protocollo *SyncML*, ad oggi conosciuto anche come *Open Mobile Alliance Data Synchronization and Device Management*, è nato negli anni 90 con l'obiettivo di garantire uno standard aperto e platform-independent per la sincronizzazione dei dati tra i primi cellulari e i computer desktop. Usato prevalentemente nella sincronia dei contatti e del calendario, in realtà SyncML si occupa solo della parte di trasporto, a prescindere dal contenuto di quanto trasportato. Per questo trova applicazioni anche in altri ambiti, come quello del backup. Per maggiori informazioni http://www.openmobilealliance.org/tech/affiliates/syncml/syncmlindex.html

ti con cui sincronizzarsi, timestamp dell'ultima sicronia effettuata ecc).

- Viene chiamanto il metodo *SyncManager.sync* passandogli *NotesSyncSource* come *SyncSource*.
- Il *SyncManager* contatta il server Funambol, si autentica e si accorda con esso su quale tipo di sincronia eseguire per il tipo di dati gestito dalla *SyncSource*.

Se non viene specificato diversamente, la prima tentata è una *TWO_WAY_SYNC*. A secondo del tipo di sincronia, il *SyncManager* trasmette al server Funambol i dati sugli elementi aggiunti, modificati e cancellati nel client, ricavati grazie ai metodi della *SyncSource* e ai paramentri *LastAnchor* e *NextAnchor*, che indicano l'intervallo di tempo all'interno del quale considerare le modifiche richieste. Il server Funambol analizza quanto ottenuto, aggiorna i suoi dati e trasmette le sue modifiche al *SyncManager*, che le persiste sul client grazie ai metodi *addItem*, *deleteItem* e *removeItem* della *SyncSource*.
Al termine della sincronia, vengono aggiornati *LastAnchor* e *NextAnchor* della *SyncSource*.
Funambol mette a disposizione un progetto chiamato client-sdk, sempre reperibile nella sezione download della Forge, contenente librerie e documentazione per usare i suoi servizi con diverse tecnologie: Java, C++ e J2ME. Al momento della scrittura di questo articolo, inoltre, è in fase sviluppo anche l'SDK per Android, presente per ora solo all'interno del progetto android-client e in grado di sincronizzare contatti e appuntamenti (https://android-client.forge.funambol.org). Per maggiori informazioni, rimandiamo al box laterare con un elenco di link a diversi documenti informativi.

NOTA

FUNAMBOL

Nata circa 10 anni fa dalla visione di un ingegnere italiano, Fabrizio Capobianco, Funambol oggi offre un prodotto di sincronia dati "on the cloud" sotto licenza Affero GPL2, comprensivo di client ed estensioni per moltissime piattaforme mobili e non. Data la natura totalmente open source del prodotto, tutto il codice sorgente è liberamente disponibile e, grazie a ciò, nel tempo si è creata una community intorno al progetto, che ha sviluppato connettori ed estensioni per ulteriori ambienti, oltre a quelli già supportati. Per maggiori informazioni www.funambol.org

CLIENT ANDROID: GESTIONE DATI

Allo scopo di semplificare il processo di sviluppo, minimizzare il codice da scrivere e concentrarsi unicamente sulle funzionalità di sincronia, abbiamo modificato ed esteso un'applicazione Android già esistente, il famoso Notepad presente nel tutorial online introduttivo alla piattaforma (http://developer.android.com/guide/tutorials/notepad). Sono state incluse le librerie del funambol-sdk per android e, direttamente dal Funambol Java SDK, sono state aggiunte alcune classi presenti nei package *com.funambol.**, che implementano l'oggetto *Note* e i metodi per la sua conversione. Successivamente, abbiamo creato una nuova attività per i impostare le configurazioni necessarie ad usare il server Funambol e lanciare una sincronia, abbiamo inserito nel menu principale una nuova voce in grado di richiamarla e abbiamo modificato lo strato di accesso ai dati per gestire più agevolmente le informazioni racchiuse nella classe *Note*.

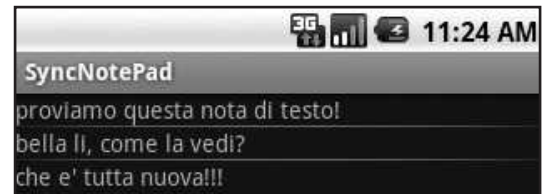



Fig. 2: L'Activity principale dell'applicazione Notepad...

Come dicevamo prima, il protocollo *SyncML* si occupa solo dello strato di trasporto, lasciando piena libertà sui dati trasportati. Per questo il server Funambol definisce un formato, il *SIF-N*, per inserire oggetti di tipo note dentro ai messaggi *SyncML*, in modo da essere indipendente da come ogni client le persiste nel suo storage locale. Per i contatti e per gli appuntamenti, invece, viene usato lo standard *VCard*. Occorrerà quindi serializzare in *SIF-N* le note ed inserirle in dei *SyncItem* prima di mandarle al server, mentre, per ogni *SyncItem* inviato da quest'ultimo al client, verrà estratto il contenuto, deserializzato dal *SIF-N* e convertito in un oggetto *Note*. I due helper che si occupano di queste operazioni sono, rispettivamente, *NoteToSIFN* e *SIFNParser*, utilizzati nei metodi di *NoteDao*. Non occorre scendere nel dettaglio di

Fig. 3: ... e quella per la creazione e la modifica di una nota

queste due classi, dato che sono prese direttamente dal client-sdk e possono funzionare per noi come una black-box. L'appena citato *NotesDao* è il nostro Data Access Object, una classe che fa da ponte tra i *SyncItem* e le informazioni grezze provenienti da *NotesDbAdapter*, occupandosi delle operazioni di conversione, gestione dei timestamp e degli status. Il metodo *getNoteFromSyncItem* viene usato per convertire il contenuto di un *SyncItem* in un oggetto *Note*: prima viene istanziato un parser passandogli il contenuto preso dal *SyncItem*, poi viene invocato il suo metodo *parse* per ottenere la *Note*. Tra le due chiamate, c'è la gestione delle possibili eccezioni generate.

```
SIFNParser parser = null;
InputStream is = new
    ByteArrayInputStream(syncItemData.getContent());
try {
    parser = new SIFNParser(is);
} catch (SAXException e) {
...
```

La conversione opposta, da *Note* a *SIF-N*, avviene nel metodo *getSyncItemFromNote*, che passa al costruttore del converter fuso orario e codifica con cui l'oggetto *Note* verrà convertito, inserendo il risultato di questa operazione all'interno di un *SyncItem*, settandone anche il tipo (*NOTE_ITEM_TYPE*) e la chiave (il progessivo della nota).

```
NoteToSIFN converter;
String convertedContent;
TimeZone tz =
        Calendar.getInstance().getTimeZone();
converter = new NoteToSIFN(tz, "UTF-8");
try {
    convertedContent =
            converter.convert(noteToConvert);
} catch (ConverterException e) {
    Log.e("getSyncItemFromNote", e.getMessage());
    return null;
}
Log.i("getSyncItemFromNote", convertedContent);
SyncItem item = new SyncItem(noteToConvert.
        getUid().getPropertyValueAsString());
item.setContent(convertedContent.getBytes());
item.setType(NOTE_ITEM_TYPE);
item.setKey(noteToConvert.getUid().getPropertyValue
        AsString());
return item;
```

Rispetto all'originale del tutorial, nel database usato per contenere i dati del client sono stati aggiunti tutti gli ulteriori campi che Outlook memorizza per ogni nota: una data ad essa associata, colore, coordinate X, Y, larghezza e altezza del riquadro che la contiene, categorie di appartenenza e folder in cui è inserita. Attualmente il nostro client non fa uso queste informazioni, ma dato che sono tutte incluse nel processo di sincronia, future implementazioni non avrebbero difficoltà a farlo. Inoltre Outlook permette di modificare solo il corpo della nota, e non il titolo. La query per creare la tabella, dentro la classe *NotesDbAdapter*, diventa quindi la seguente:

```
private static final String DATABASE_CREATE =
        "create table notes (" +
            "_id integer primary key autoincrement, " +
            ...
```

Come si può notare, sono stati aggiunti anche due campi per le informazioni funzionali al processo di sincronia: status dell'elemento e data di ultimo aggiornamento. Tra i diversi valori che lo status può assumere, ci sono quelli di *NEW*, *UPDATED* e *DELETED*, mentre *last_update* contiene il timestamp di questo cambio di status. *NoteSyncSource* deve infatti sapere quali sono le note nuove, modificate o cancellate in un range temporale, generalmente dall'ultima sincronia al momento corrente, e per farlo si appoggia *NoteDao*, sfruttando questi due campi. La seguente query, ad esempio, permette di trovare le note create successivamente all'ultima operazione di sincronia andata a buon fine:

```
StringBuilder where = new StringBuilder();
where.append(KEY_LASTUPDATED).append(">").appe
                                    nd(since)
    .append(" AND ")
.append(KEY_LASTUPDATED).append("<").append(to
                                            );
    .append(" AND ")
.append(KEY_STATUS).append("='").append(status).a
                                     ppend("'");

Cursor mCursor =
    mDb.query(DATABASE_TABLE,
            ALL_TABLE_COLUMNS,
            where.toString(),
            null, null, null, null);
return mCursor;
```

status contiene il valore *SyncItemState.NEW*, che indica una nuova nota, mentre le variabili *since* e *to* contengono due timestamp, rispettivamente quello della data dell'ultima sincronia e quello del momento corrente. Sono entrambi memorizzati nel client grazie alla classe *NotePreferences* come campi *LastAnchor* e *NextAnchor*. La seconda viene inviata dal server all'inizio di ogni operazione di sincronia e, se questa va a buon fine, viene memorizzata in *LastAnchor* al suo termine. Allo scopo di mettere in

FUNAMBOL FORGE

Funambol Forge è un repository di progetti in stile SourceForge che ospita sia i progetti sviluppati e manutenuti da Funambol, sia quelli creati e gestiti dalla community. La home page contiene informazioni generali su Funambol con accesso diretto alle pagine di download dei vari componenti, server e client, documentazione e altro. Il progetto Core e i sottoprogetti contengono la maggior parte del codice di Funambol Community Edition. Sul Forge sono anche ospitate le pagine per i programmi di partecipazione alla community Funambol, come i Code Sniper, Phone Sniper e L10n Sniper: occasioni per entrare in contatto con questo mondo, con la sicurezza di essere retribuiti per il lavoro fatto.

grado la *SyncSource* di conoscere le note rimosse sul client dall'ultima sincronia, abbiamo scelto di eliminarle in modalità *soft-deletion*, cioè senza cancellare fisicamente il record corrispondente alla nota, ma impostando a 'D' il suo status e aggiornando il timestamp presente in *last_update*. Coerentemente con questa politica di soft-deletion, modifichiamo il metodo *NoteDbAdapter.fetchAllNotes*, introducendo il seguente controllo sullo status di un record del DB:

```
public Cursor fetchAllNotes()
{
    return mDb.query(DATABASE_TABLE,
                new String[] {KEY_ROWID, KEY_TITLE,
                                    KEY_BODY},
        KEY_STATUS + "<>'D'",
        null, null, null, null);
}
```

Ricordiamoci comunque che, forzando nel *NoteSyncSourceConfig* una sincronia di tipo *REFRESH_FROM_SERVER*, tutte le note presenti sul client vengono fisicamente cancellate. Questo tipo di sincronia va usato per riportare sul client quanto contenuto sul server, senza che il primo possa trasmettere le ultime modifiche a quest'ultimo. Si può anche impropriamente usare una *REFRESH_FROM_SERVER* per fare un po' di pulizia e liberare risorse sul client, sempre preziose, se si creano e cancellano note in modo intensivo.

CLIENT ANDROID: LA SINCRONIA DATI

Passiamo ora alla creazione dell'elemento più importante della sincronia: la classe *NoteSyncSource*. In fase di inizializzazione, le viene passata un'ulteriore classe, *NoteSyncSourceConfig*, che contiene le configurazioni minime di cui la *SyncSource* necessita:

```
public NoteSyncSourceConfig(Activity a) {...}
```

Sono fondamentali la *RemoteUri*, che indica il database su server Funambol da utilizzare per sincronizzare i dati, quello delle note appunto, e il *Type* dei dati contenuti all'interno del *SyncItem* gestito dalla *SyncSource*. Nel nostro caso, come avevamo spiegato, viene usato il SIFN, mentre per i contatti avremmo trovato un *Type* uguale a *text/x-vcard*. Non viene applicato nessun tipo di *Encoding* al contenuto del *SyncItem*. Troviamo anche l'Activity che serve per ottenere l'Application Context necessario alla *SyncSource* per molte delle sue operazioni. Gli altri medoti di *NoteSyncSource* sono abbastanza semplici e quasi sempre corrispondenti ad un metodo su *NoteDao*. Ad esempio, l'inserimento di una nuova nota arrivata dal server chiamerà il metodo *NoteSyncSource.addItem*

```
@Override
public int addItem(SyncItem item)
    throws SyncException
{ ... }
```

A sua volta, superato il controllo della possibilità di aggiungere un elemento per il tipo di sincronia corrente, *NoteSyncSource.addItem* chiama il medoto *NoteDao.addNote*

```
public int addNote(SyncItem item, int key) {... }
```

Come si legge dal codice, il valore di ritorno del metodo *NoteSyncItem.addItem* viene impostato su 200 quando un'operazione va a buon fine, secondo quanto dettato dallo standard *SyncML*. Valori come *500*, ad esempio, indicano invece diversi tipi di errori. Rispetto al programma Notepad originale, è stata aggiunta nella nostra applicazione una nuova *Activity, FunambolSettings*, che permette di specificare username, password e server sul quale effettuare la sincronia, oltre che ad esporre una TextView contenente i messaggi. Il log delle singole operazioni della sincronia è reso possibile da *NoteListerner*, una classe derivante da *SyncListener* che viene agganciata a *NoteSyncSource* in fase di creazione di quest'ultima. Il meccanismo di funzionamento è semplice, dato che la *SyncSource* chiama i diversi

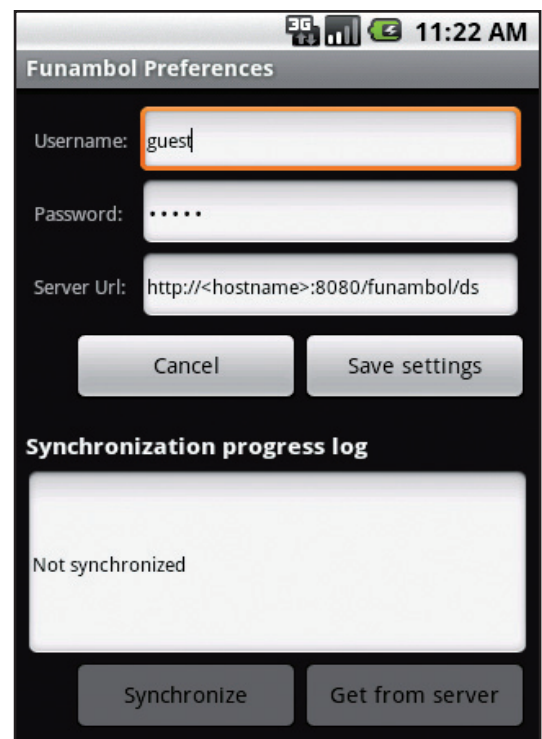



Fig. 4: L'Activity che abiamo aggiunto ci consente di impostare username, password e server da usare per sincronizzare le nostre note...

MYFUNAMBOL

MyFunambol (http://my.funambol.com) è il portale, liberamente utilizzabile, per provare le funzionalità offerte dalla piattaforma Funambol. Una volta registrati, si può scaricare il client adatto al tipo di smartphone posseduto e lanciare la prima sincronia. Oltre alla possibilità di gestire i contatti, note e appuntamenti via web, il portale può essere usato come semplice backup dei dati del propro dispositivo, funzionalità utile in caso di perdita o sostituzione del device. Se si è sempre alla ricerca di funzionalità borderline, all'indirizzo http://dogfood.funambol.com è raggiungibile la versione di beta-testing dello stesso portale, con implementate le ultimissime funzionalità da sperimentare. Attualmente, ad esempio, è possibile assegnare ai propri contatti la foto del loro profilo su Facebook.

metodi del listener in base alla tipologia di operazione svolta. Questi metodi, implementati nel nostro *NoteListerner*, non fanno altro che aggiornare la *TextView* con i log, accodando il nuovo messaggio.

Fig. 5: … ed ecco il log della nostra prima sincronia!

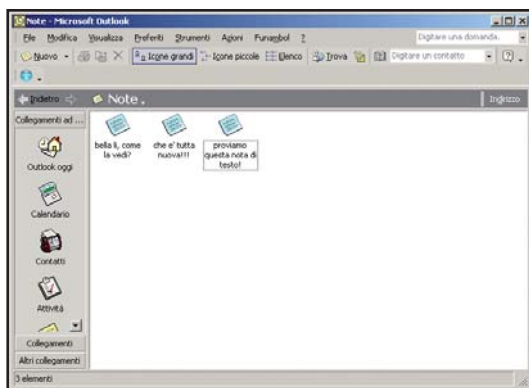



Fig. 6: Le note sono state trasferite su Outlook con successo

CLIENT ANDROID: GLI ALTRI COMPONENTI

Appoggiandosi alla classe *BaseSettings*, alcune informazioni inerenti la *SyncSource* e il *SyncManager* vengono memorizzate ricorrendo al meccanismo delle *Preferences*. Si tratta di una pratica alternativa alla creazione di file o tabelle per conservare informazioni proprie dell'applicazione, che devono essere mantenute tra un lancio e l'altro. Funziona così: si memorizzano i dati interessati in una lista di chiavi-valori (l'oggetto *SharedPreferences*) per poi chiamare i metodi del *Context* che permettono di caricarlo e di persisterlo per mezzo dell'oggetto *SharedPreferences.Editor*. Ogni *Context* può contenere molteplici oggetti *SharedPreferences*, ognuno di essi identificato da un nome univoco. Nella nostra applicazione, abbiamo usato l'identificativo *fnblPref*. Di seguito un classico esempio di creazione di un container per le preferenze: memorizzazione di un valore di tipo stringa facente capo alla chiave "*syncUrl*" e lettura di quest'ultimo.

```
public static void getAndSetPreference(Activity a) { ... }
```

L'*Activity* corrente viene usata per ricavare il riferimento al *Context* dell'applicazione. È chiaro che questo metodo è utile per piccoli e semplici insiemi di dati, mentre per qualcosa di più articolato e complesso occorrerà rivolgersi al altre tecniche di storage persistente dei dati. Nell'*Activity* che si occupa della sincronia, in base al bottone premuto, viene richiesto un diverso tipo di sincronia: una *SyncML.ALERT _CODE_REFRESH_FROM_SERVER* che forza una cancellazione dei dati del client e prende in carico solo quelli presenti sul server, oppure una *SyncML.ALERT_CODE_FAST*, che tenta invece di eseguire la classica *TWO_WAY_SYNC*, operazione in cui client e server si scambiano solo le modifiche dall'ultima sincronia terminata con successo. Seguendo le linee guida per lo sviluppo di applicativi, questo blocco viene gestito come thread separato rispetto all'applicazione principale, in modo da non bloccarla e da poter mostrare un progessivo dei log di quanto viene eseguito. Per raggiungere lo scopo, è stata creata la classe privata *SyncThread*, che estende la classe *Thread* e che all'interno del metodo *Run* ha il codice che lancia la sicronia vera e propria. Nel seguente frammento di codice possiamo vedere l'inizializzazione della *NotesSyncSourceConfig*, alla quale viene passata l'*Activity* corrente e il tipo di sincronia richiesta, l'inizializzazione della *NotesSyncSource*, alla quale viene passato l'oggetto con le configurazioni della sincronia appena istanziato e l'*Activity* corrente, l'inizializzazione del *NotesListener*, al quale viene passato l'*EditText* che conterrà i log e, infine, il collegamento del *Listener* alla *SyncSource*:

```
NoteSyncSourceConfig noteconf = new
        NoteSyncSourceConfig(FunambolPreferences.this,
                syncMode);

NoteSyncSource src = new NoteSyncSource(noteconf,
                FunambolPreferences.this);
src.setListener(new NoteListener(Funambol
                Preferences.this, mHandler, txtLogArea));
```

Alfredo Morresi

L'INDIRIZZO DEL SERVER

Se si sta usando la versione online di Funambol, l'URI del server con cui sincronizzare i dati sarà http://my.funambol.com/sync, mentre se si sta usando quella in locale, l'URI sarà http://indirizzo_ip_locale:8080/funambol/ds, dove *indirizzo_ip_locale* sarà l'indirizzo del pc, raggiungibile dal telefono Android.

Questo approfondimento tematico è pensato per chi vuol imparare a programmare e creare software per gli smartphone con sistema operativo Google Android. La prima parte del testo guida il lettore alla conoscenza degli strumenti necessari per sviluppare sulla piattaforma mobile di Mountain View (installazione SDK, librerie e tool di supporto allo sviluppo).
Le sezioni successive sono pensate per un apprendimento pratico basato su esempi di progetto: dialogo e interazione con l'ambiente operativo del telefonino, interazione con gli utenti, componenti di un widget, interfacce in XML, gestione del touch, progettazione dei menu e via dicendo.
Una serie di esempi pratici da seguire passo passo che spingono il lettore a sperimentare sul campo il proprio livello di apprendimento e lo invitano a imparare divertendosi.

www.punto-informatico.it